MASTER'S THESIS
COMPUTER SCIENCE AND MEDIA

Developing a Fire Propagation System for a Massively Multiplayer Online Game

---

submitted by

Michael MUELLER
Matr.-Nr.: 32725

at Stuttgart Media University
on June 18, 2018

in fulfillment of the requirements
for the degree of Master of Science

Supervisor:
Prof. Dr. Stefan RADICKE
Stuttgart Media University

Co-Advisor:
Nikolay STEFANOV
Massive Entertainment

Eidesstattliche Erklärung

Hiermit versichere ich, Michael MUELLER, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit (bzw. Masterarbeit) mit dem Titel: "Developing a Fire Propagation System for a Massively Multiplayer Online Game" selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht.
Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.
Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

_____ _____

Ort/Datum Unterschrift

Abstract

Massively Multiplayer Online Games (MMOGs) are increasing in both popularity and scale. One of the reasons for this is that interacting with human counterparts is typically considered much more interesting than playing against an Artificial Intelligence. Although the visual quality of game worlds has increased over the past years, they often fall short in providing consistency with regard to behavior and interactivity. This is especially true for the game worlds of MMOGs. One way of making a game world feel more alive is to implement a Fire Propagation System that defines how fire spreads in the game world. Singleplayer games like *Far Cry 2* and *The Legend of Zelda: Breath of the Wild* already feature implementations of such a system. As far as the author of this thesis knows, however, no MMOG with an implemented Fire Propagation System has been released yet. This work introduces two approaches for developing such a system for a MMOG with a client-server architecture. It was implemented using the proprietary game engine *Snowdrop*. The approaches presented in this thesis can be used as a basis for developing a Fire Propagation System and can be adjusted easily to fit the needs of a specific project.

Kurzfassung

Die Popularität und der Umfang von Massively Multiplayer Online Games (MMOGs) steigt. Einer der Gründe dafür ist, dass die Interaktion mit einem menschlichen Gegenüber in einem Spiel als sehr viel reizvoller wahrgenommen wird als mit einer künstlichen Intelligenz. Obwohl die visuelle Qualität von Computerspiel-Welten im Laufe der letzten Jahre zugenommen hat erreicht sie oft nicht dieselbe Qualität in Bezug auf Verhalten und Interaktivität. Dies gilt besonders für Computerspiel-Welten in MMOGs. Ein Fire Propagation System, welches die Art und Weise, wie sich Feuer in der Computerspiel-Welt ausbreitet, definiert, ist eine Möglichkeit um eine solche Welt lebendiger wirken zu lassen. Singleplayer Spiele wie *Far Cry 2* und *The Legend of Zelda: Breath of the Wild* können bereits die Implementierung eines solches Systems vorweisen. Nach dem Kenntnisstand des Autors wurde jedoch noch kein MMOG mit solch einem System veröffentlicht. Diese Arbeit zeigt zwei Ansätze für ein solches System in einem MMOG mit einer Client-Server Architektur. Das System wurde mit der Game Engine *Snowdrop* umgesetzt. Beide hier vorgestellten Ansätze dienen als Basis für ein Fire Propagation System und können an projektspezifische Bedürfnisse angepasst werden.

Contents

List of Figures

List of Functions

List of Abbreviations

AABB	**A**xis-**A**ligned **B**ounding **B**ox
AI	**A**rtificial **I**ntelligence
MMOG	**M**assively **M**ultiplayer **O**nline **G**ame
MMORPG	**M**assively **M**ultiplayer **O**nline **R**ole **P**laying **G**ame
RTT	**R**ound **T**rip**T**ime
UID	**U**nique **Id**entifier
UML	**U**nified **M**odeling **L**anguage

Chapter 1

Introduction

Video games are the largest segment in entertainment and the market is growing fast [25]. With more than 125 million users, Steam is the largest digital distribution platform [7, 28]. The number of games released on Steam has increased over 400% in the last four years [8] with an average playtime remaining stable over that period [11, 12, 13, 14]. To stay competitive in this fast growing market, game publishers focus on live service games, extending the life cycle of a game while keeping players more engaged [25].
Steam's statistics suggest that it is particularly games offering a multiplayer experience that are very popular among gamers. Of the five most played games on Steam, three are multiplayer-only experiences while all of them provide online functionalities [29]. Massively Multiplayer Online Games (MMOGs) are a popular form of online games, representing 60% of all digital PC revenue [23]. They allow a large number of players to share a single game world.

1.1 Motivation

While the visual quality of MMOGs is increasing, the way players interact or influence the game world and its objects, although equally important, has not changed much. Research results suggest that the lack of coherence between objects behaving in a realistic way and the visual representation of the game world breaks player immersion [15]. *Fractured*, a first-person open-world Massively Multiplayer Online Role Playing Game (MMORPG) in development claims to be the first of its kind to mix "action combat with fully interactable environments" [6]. In contrast to this the use of dynamic systems in offline singleplayer open-world games has increased. A modern example of this is *The Legend of Zelda: Breath of The Wild* [21]. By implementing a *Chemistry Engine*, the developers were able to create a much more dynamic world with many possible ways of interaction between objects and players [5]. For example, by adding the element of water to the game world, rain makes it much harder for the player to climb rock faces, as they become slippery when they are wet. Furthermore, the player cannot set anything on fire while it is raining.

One of the more common dynamic systems implemented in modern games like *Assassins Creed Origins* or the *Far Cry*–Series since *Far Cry 2* is the propagation of fire. In most games players are not able to interact with the element of fire. As there is no dynamic system behind it, fire is static and does not propagate within the game world. In cooperation with Massive Entertainment, this thesis presents two approaches for a Fire Propagation System in an MMOG.

1.2 Objective and Scope

Thee two solutions offered in this thesis to implement a Fire Propagation System were developed for an MMOG in pre-alpha status. The objective of this thesis is to provide readers with a genre-independent approach for developing a Fire Propagation System in an MMOG. Both approaches are realized as prototype and were developed in Massive Entertainment's proprietary game engine Snowdrop [18]. The implementation and the concept are presented in a generic way through Unified Modeling Language-diagrams (UML) and pseudo-code that can be adapted for any MMOG-project. One of the approaches is supported by an example of interaction with another gameplay system. The scope of this thesis is limited to the description of the way Fire Propagation Systems function for in MMOGs and only provides a brief overview of the technology behind MMOGs.

1.3 Related Work

This chapter provides an overview of technologies and games that share aspects with the objective of this thesis. None of the presented works implement a Fire Propagation System in the context of a MMOG, but they all provide an foundation to build such a system.

Change and Constant: Breaking Conventions with *The Legend of Zelda: Breath of the Wild*

With the development of a *Chemistry Engine*, Nintendo introduced the concept of Materials and Elements in *The Legend Of Zelda: Breath Of The Wild* creating a „rule-based state calculator“ [5]. By following three simple rules, they were trying to achieve what they called „Multiplicative Gameplay“. Transferring the concept of chemical reaction into a gameplay system.

1. ***Elements* can change a *Material's* state**
   Elements like fire, water and ice, that don't hold a constant form are able to change the state of Materials, like trees, rocks and even the player, that are solid objects.

2. ***Elements* can change another *Element's* state**
   Elements are able to change each others state.

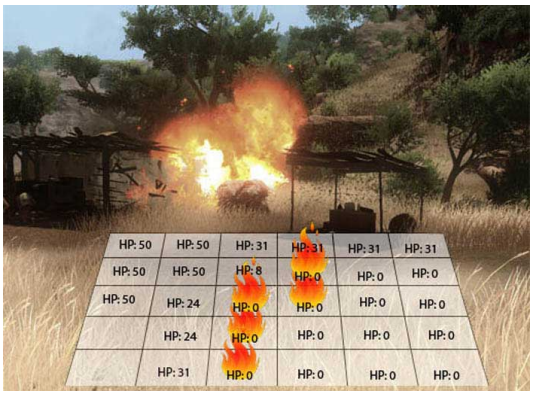

FIGURE 1.1: *Far Cry 2* fire grid

3. ***Materials* can't change another *Material's* state**
   Two solid objects can't change each others state. Only the interaction with an Element results in a change of a Material's state.

On a basic level these rules also apply to how a fire propagation system works. Flammable objects are defined as *Material* and fire as *Element*. Although being an open world game, *The Legend Of Zelda: Breath Of The Wild* is an offline singleplayer experience and doesn't feature any kind of online functionality.

Far Cry: How the Fire Burns and Spreads

The Fire Propagation System developed for *Far Cry 2* is described in the blog post *Far Cry: How the Fire Burns and Spreads* [16]. The developer used a 2D grid for grass and a 3D grid for larger objects like trees as an underlying base structure. Each grid cell has a position, a radius and hit points. If the amount of hit points of a cell reaches zero, the cell is considered to be on fire and starts damaging adjacent cells during its *Burning Lifetime* (Figure 1.1). Cells in wind direction receive more damage, simulating a faster spreading fire in wind direction. To limit the spreading of fire, each fire source has a limited amount of *Spreading Points* that are consumed by reducing the hit points of a grid cell. When a cell is set on fire, it sends out an event to other systems in the surrounding area, enabling the Artificial Intelligence (AI) of the game to react on objects that are on fire. While this system makes fire behave in a, for the player, believable way, it was also being developed with an offline single player experience in mind.

SpatialOS

With the cloud based game platform *SpatialOS* [10], Improbable Worlds Limited tries to enable developers to create a big, persistent, dynamic world shared by thousands of players. By letting the simulated world be handled by many servers, they promise „huge, seamless worlds" that can handle „a huge number of concurrent players" [10]. Each server, or *Worker*, only knows about a part of the game world (Figure 1.2).

FIGURE 1.2: SpatialOS structure

This way, servers only have to handle a limited amount of objects. This technology aims to eliminate the performance constraints of running an game world on a single server.

1.4 Structure of this Work

Chapter 2 provides some background on MMOGs, their topology and how the state of the shared game world is synchronized. Furthermore, basic principles of Fire Dynamics are explained. This includes heat, temperature, heat transfer and Fire Development. Moving on to Chapter 3, an overview of important concepts and systems of Snowdrop is given, while this Chapter is not concerning itself with the implementation, it is important for some aspects of the presented approaches. Chapter 4 then lists the defined requirements before going over both approaches, starting with the concept, followed by the implementation and identified limitations. An evaluation where both approaches are evaluated and compared is presented in Chapter 5. Finally, Chapter 6 provides a conclusion and gives an outlook of future work.

Chapter 2

Background

2.1 Massively Multiplayer Online Games

Within MMOGs a persistent, virtual world is featured for interaction between large amounts of players in real-time. The term MMOG does not specify the genre of the game. In fact, the concept of a MMOG has been combined with several genres:

- MMORPG: Massively Multiplayer Online Role Playing Game
- MMOFPS: Massively Multiplayer Online First Person Shooter
- MMORTS: Massively Multiplayer Online Real Time Strategy

MMOGs are almost exclusively built around a client-server architecture, where the state of the virtual world is updated on the server and sent to all connected clients [4]. Additionally clients are only trusted to send interaction requests and receive updates from the server to prevent player cheating [19]. Because of that reason both presented approaches in this thesis are based on the assumption that the MMOG was built with a Client-Server topology.

2.1.1 Client-Server Topology

The connection of computers is determined by its network topology. In the context of a MMOG the organization of participants and their game state synchronization are defined by network topology.In a Client-Server Topology one of the connected game instances is the designated server, while the others are the designated clients. The server is determined to be responsible for communicating with all clients. A client only communicates with the server. This is illustrated in figure 2.1. Based on this there are $O(2n)$ active connections in each Client-Server Topology, $O(n)$ connections from the server to n clients and $O(1)$ connections for each of the n clients to the server. Due to increasing numbers of connections, the bandwidth of the server linearly increases. The server needs to be prepared to handle b bytes per second for each client, resulting in $b * n$ bytes per second for n clients. If this topology is implemented, an authoritative server is mostly used. In that case the game instance running on the server will be accepted by every client even if their local game instances are in a different state. Every interaction of a client

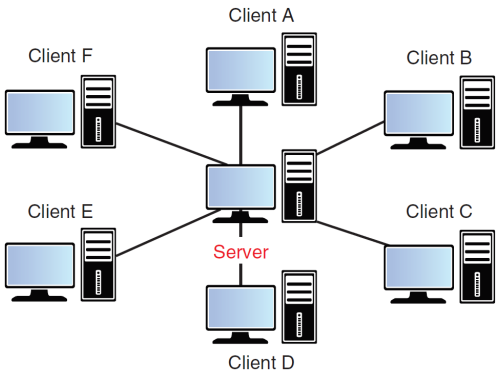

FIGURE 2.1: Client-Server Topology [9]

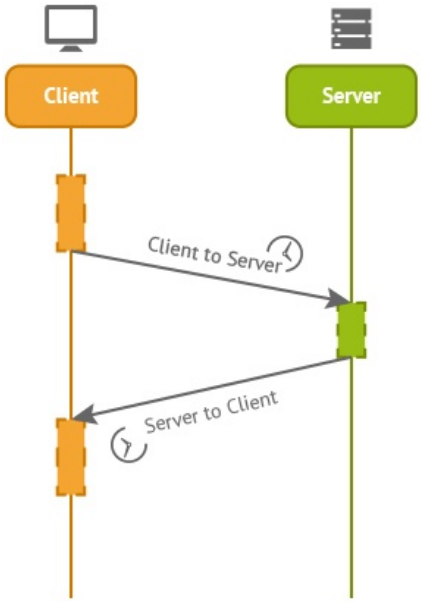

FIGURE 2.2: Round Trip Time [30]

that changes the game state needs to be approved by the server. Afterwards all affected clients will be provided with the result of this action. The client itself is not allowed to make a decision about what happens after the action. Implementing an authoritative server prevents player cheating, as local changes need to be verified by the server before other clients receive updates. This also introduces some amount of lag, because every action needs to be verified by the server before it can be executed as seen in figure 2.2. The lag results from the amount of time needed for "packets to travel to and back from a particular computer in the network."[9]. This is also called Round Trip Time (RTT). [9]
In the example, shown in Figure 2.3, the RTT from Mary to the server is much longer than from John to the server. Even though the *shoot* - event from Mary is sent to the server earlier, John's *shoot* - event is received earlier. Thus the server calculates that Mary was hit first. As a sub-classification of servers, dedicated servers, only run the game state and the running process is completely separated from the clients processes running the game. This means that the code executed on the server

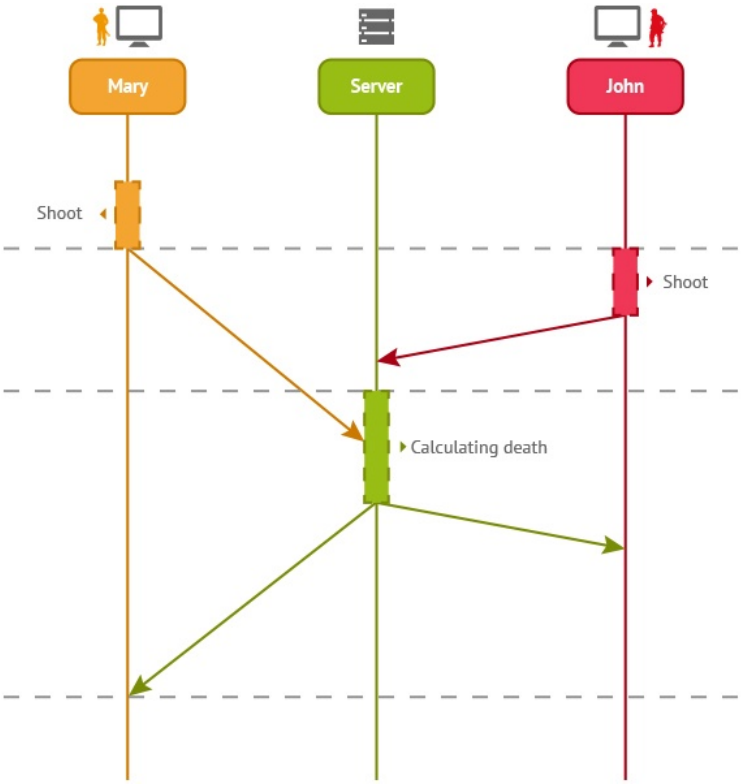

FIGURE 2.3: Lag and Round Trip Time [30]

is different from the code executed on each client. There might be properties or functions that are needed on the server, but not on the client. Inheritance and virtual functions are essential to achieve this kind of behavior. The client and the server version of a class (the derived classes), might share the same functionality (of the base class), but override or implement new member functions.[9]
The process running on the server does not display any graphics and allows developers to run multiple dedicated server processes on a single machine.
Another sub-classification of servers is the listen server, where the server is actively participating in the game. In this case, the server happens to be hosted by a player in the game.[9]
Contrary to the client-server topology, all participants (peers) are connected with each other in the peer-to-peer topology. This means, with n clients, there are O(n^2) connections.[9]
A common approach to keep the game state in sync between the peers is to share inputs of each peer across all other peers. For this to result in the exact same state on each peer, the game implementation "needs to be fully deterministic".[9]

2.1.2 Game State Synchronization

For a game in which players share a game world, players also need to feel like they are playing in the same world. Each player action needs to be replicated on all clients

for the world state to remain consistent. Only changes to the world state have to be exchanged. Therefore it is not necessary to replicate the entire world state.
GLAZER defines the world state as the state of all game objects (entities) in a world, assuming an object-oriented game object model [9]. As such to maintain consistency of the world state for each client, the state of each entity needs to be replicated. For *Halo: Reach* [1] the state of an entity consists of:

- Position
- Health
- around 150 further properties

that vary depending on the type of game object. ALDRIDGE [1] highlights that this *State Data* will eventually be replicated, but there is no guarantee of how many intermediate states are being sent. The number of these intermediate states varies depending on the available bandwidth. This represents one of the three replication actions defined by GLAZER [9]. The other two replication actions are:

- Create game object
- Destroy game object

State Data itself does not provide information about why there was a state change. For this, *Events* provide additional data. *Events* are "unreliable notifications of transient occurrences" [1], describing transitions. While the final state is guaranteed to be sent with *State Data*, *Events* are only sent when the state transition is visible to the player. An example for this could be the death of a player character, which is replicated by *State Data*. But when other players can not see this character, the *Event*, that can be the firing of a weapon, is not sent. As mentioned in 2.1.1 an authoritative server requires client actions to be approved before they are executed and replicated. These requests are sent to the server in form of *Events*. Lastly, *Control data* contains player control inputs. This is sent as frequently as framerate and bandwidth of the server allows and improves the accuracy of predicting player movement based on control input.

2.2 Fire Dynamics

As the presented approaches in this thesis are loosely based on physical concepts of heat and temperature, this section provides an overview of how these concepts influence fire behavior.

2.2.1 Heat and Temperature

Heat is the amount of energy that is flowing from a warmer object to a colder one. The thermal energy of a system is being passed to another system when there is a

temperature difference. Thermodynamics is dealing with heat and temperature. The four laws of thermodynamics are employed by any thermodynamic system. Only the relevant laws for this thesis will be listed:

1. First law of thermodynamics:
   In a completely insulated system no thermal energy (heat) is lost or gained regardless of the way the heat diffuses. The total energy of the system must stay the same.

2. Second law of thermodynamics:
   In a completely insulated system the temperature inside the system must finally become the same everywhere, regardless of which way the heat diffuses.

2.2.2 Heat Transfer

Heat transfer is the rate of thermal energy being transferred from one system to another at any instant due to temperature difference. This occurs through three ways:

1. conduction, the heat transfer trough direct contact of two bodies,

2. convection, the heat transfer through fluid flow,

3. radiation, the heat transfer through the propagation of light.

In the context of this thesis convection and radiation are not considered during the development of the approaches.

Thermal Conduction

Thermal conduction is the transfer of heat via direction contact of two bodies. The heat flow is happening within and through the bodies themselves. Fourier's Law of Heat Conduction states the following relationship:

$$\frac{\Delta Q}{\Delta t} = -k * A * \frac{\Delta T}{\Delta x} \tag{2.1}$$

where:

$\frac{\Delta Q}{\Delta t}$ is the thermal energy (ΔQ) transferred per unit time (Δt)
k is the thermal conductivity of the material/object
A is the conduction area
$\frac{\Delta T}{\Delta x}$is the temperature difference (ΔT) along the distance (Δx) in direction of conduction.

[24]

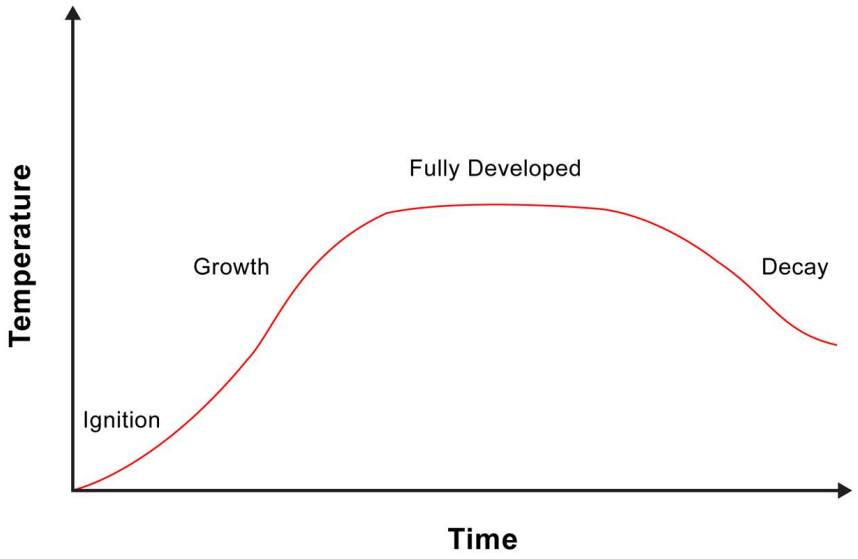

FIGURE 2.4: Fire Development Curve [20]

2.2.3 Fire Development

Fire Development is influenced by factors like fuel (properties and quantity), ventilation, compartment geometry, location of fire and ambient conditions (temperature, wind). It consists of four stages: Ignition, Growth, Fully Developed and Decay.
In the Fire Development Curve 2.4 the fire development for a fuel limited fire over time is shown.As more fuel becomes involved in the fire, the energy level increases. This process continues until all available fuel is burning (fully developed). The energy level decreases, as fuel is burning away. [20]

Chapter 3

Framework Overview

This chapter contains an overview of the framework. General concepts that are valid for the entire development environment are explained. Furthermore, systems that are implemented in *Snowdrop* [18] and are used for the presented approaches are described.

3.1 Entity-Component-System

Entity-Component-System is a pattern commonly used in game engines, like *Unity* [26]. It is based on the principle of composition [22].

3.1.1 Entity

An entity is an empty shell used as container for components [22]. In some cases an entity's only property is an unique identifier (UID) [2] or it requires a certain component to be added [27]. In the context of this project the definition of *BorealGames* [2] is used. "The entity is an implicit aggregation of the components tagged with its UID". It is also assumed, that every entity contains a position, rotation and scale.

3.1.2 Component

Components are "reusable packages" [22] of data. They add properties and behavior to entities and can be combined to create complex entities. In most cases, components, even when added to the same entity, don't know about each other. This enables parallel work on multiple components. In *Snowdrop* components only know about the entity they have been added to. Furthermore, they usually do not implement any methods. Following this pattern the implementation of the Fire Propagation System only added accessors functions to all Components.

3.1.3 System

A system runs continuously on a global level. It performs global actions on entities with a component "of the same aspect" [17] as the system. Systems provide the method-implementation for components, as they do not implement any methods themselves. For `n` types of components, there are `n` systems.

3.2 Multi-threaded Environment

Snowdrop is a multi-threaded game engine. According to this shared data can be simultaneously accessed by multiple threads. This happens parallel by executing different functional blocks and tasks to utilize all available processors. These tasks are executed asynchronously on multiple threads. A thread pool is managed by the first started threads. It is launching tasks and joins threads from the thread pool. For shared resources mutual exclusions must be enforced by the acquisition of locks. By defining critical sections in code, simultaneous access to shared data by multiple threads can be prevented. For this mutual exclusions are essential. While this prevents race conditions or deadlocks, it can produce bugs. In the context of this thesis mutexes were used as a synchronization primitive to protect shared data.

3.3 World Model

The *World Model* class represents the sum of all systems, entities and components of a single game world instance. World Models exist on three different levels: on a global, a server and a client level. They all inherit functionality of the *World Model* base class, but implement methods differently and vary in additional functionality, as well as the number of systems, entities and components.

Global

For every dedicated server process running, only one *Global World Model* exists. Client processes do not hold *Global World Models*.
It holds data and functionality that is shared among all players in one shared game world. This data exists as long as the *Global World Model* instance is loaded. The *Global World Model* stores all existing *Server-* and *Client World Models*.

3.3.1 Server

The *Server World Model* runs on the dedicated server process as well, but other than the *Global World Model*, it is created per client. It represents the world instance of a single client on the server.

3.3.2 Client

The *Client World Model* is a replication of the instanced client world on the server, with exceptions. Some systems or entities exclusively exist in either the *Server-* or the *Client World Model*.
It is created per client and does not exist on the server.

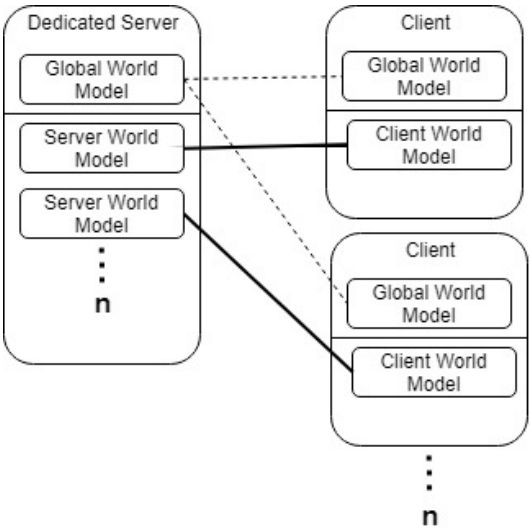

FIGURE 3.1: WorldModel Relation

3.4 Sectors

Entities are organized in a spatial index grid structure. A spatial index is called sector. A sector covers a predefined area of the game world and includes all entities and their components within this area. All sectors are of equal size. As the player moves through the game world, sectors in the view frustum of the player are loaded. Sectors close to the player are loaded while sectors far away are being unloaded. This results in a constant entity loading and unloading during a play session. These distances are predefined. Additionally the view frustum culling prevents sectors being loaded whose entities are not visible for the player at his current position.

3.5 Entity States

Entity States are defined by the `EntityState` enumeration on a global level. Both presented approaches incorporate Entity States.These states are:

1. `Default`
2. `OnFire`
3. `BurntDown`

This list presents the sum of all states that are used by the object-based and the cell-based approach.

In the context of *Snowdrop*, Entity States are not an implementation of the State Design Pattern. The `EntityState` enumeration is a member of the `EntityStateComponent`, which defines the Entity State for an entity. If not stated otherwise, the term *state* always refers to the Entity State. This allows systems to execute different behavior based on the current state of the entity and thus enabling

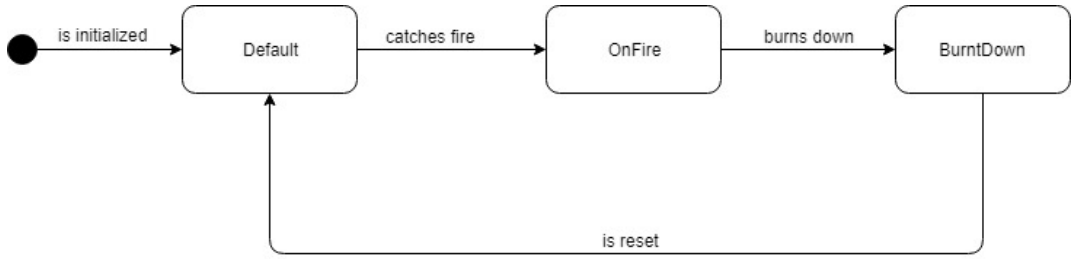

FIGURE 3.2: State Transition

other systems to implement the State Design Pattern. An entity can be in multiple states at the same time. All Entity State transitions have to be handled by systems using them. The `EntityStateComponent` only provides functions to manipulate the current state.

3.5.1 State Transition

All possible state transitions are independent from the presented approaches. Although technically possible, an entity within the Fire Propagation System can never be in two states at the same time. In the State Transition Diagram 3.2 all possible transitions are shown. Each entity is initialized with the state Default. From this state only a transition to `OnFire` is possible. Entities in the state `OnFire` inevitably switch to the state `BurntDown` after a specific period of time defined in the Fire Propagation System. From the `BurntDown` state, entities can go directly into the Default state again. A direct transition to `OnFire` from `BurntDown` is not possible. An `EntityStateComponent` instance is not automatically added to each entity in the game world. Non-interactive entities for example do not need an `EntityStateComponent` instance attached to them, because they never change their state.

3.6 SpatialGrid

The spatial hash is a hash table with three-dimensional vectors as keys and a list of classes as value. Each key-value pair is referred to as cell. The `SpatialGrid` class is an implementation of this structure. The class stored in each cell is implemented as a class template implementing two functions:

- `GetPosition()`
- `GetRadius()`

A class stored within the spatial hash is required to have a position and a radius. The `SpatialGrid` class globally defines a radius for all cells. Based on these parameters, it is determined which cells each class instances is inserted in. A class instance can be inserted in multiple cells, but not in the same cell twice. In figure 3.3 this process

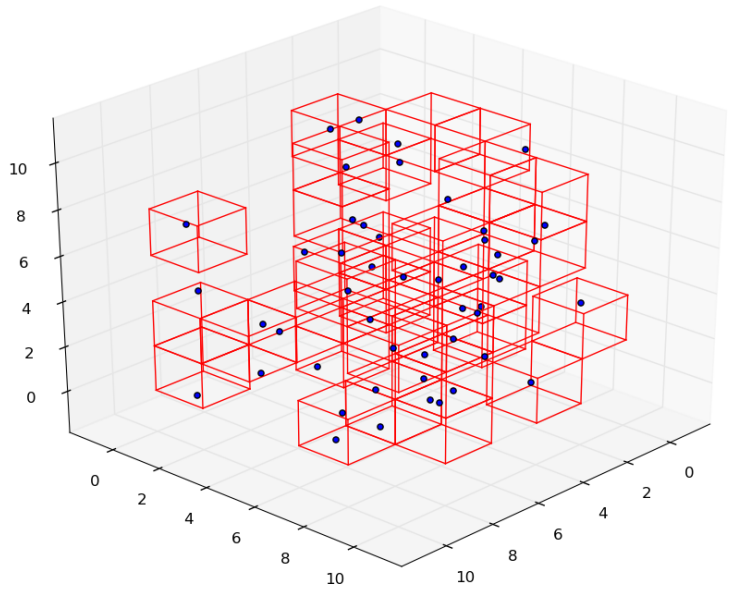

FIGURE 3.3: Spatial Hashing [3]

is visualized. Each red cube represents a cell in the spatial hash, while blue dots represent class instances. In the `SpatialGrid` class implementation of this concept, these blue dots can vary in size but not in shape. All inserted class instances are represented by spheres of different sizes.

Chapter 4

Fire Propagation System

The development of this Fire Propagation System is inspired by the Fire Propagation System of *Far Cry 2* 1.3.
In chapter 4 the requirements that are set in place for the system are explained. Furthermore, an explanation of both concepts is given. The implementation details are presented as UML diagrams and pseudo code. Additionally identified limitations are discussed.

4.1 Requirements

For the realization several requirements have to be set in place. These requirements are formulated in a generic way as the system needs to be able to work with any MMOG-project. Each requirement has its unique number and will be discussed in Chapter 6.

1. **Fire spreads in a way within the game world that players expect**

   The propagation of fire needs to behave in a way, players expect it to behave. Providing meaningful visual and auditive feedback for the player is more important than incorporating more factors that influence the propagation.

2. **Other systems need to be able to react on the propagation of fire**

   The AI in a MMOG should be able to react on objects being on fire. It must be able for other systems to create a hook and use data of the propagation system.

3. **The system has to be optimized to run for 500 clients connected to a single dedicated server**

   A server instance has to run 500 concurrent game worlds. Each player can create multiple fires at the same time within one world. This is resulting in potentially thousands of fires the server has to simulate. A target frame rate of 10 frames per second for the server process has to be achievable with this system.

4. **There should be no frame time spikes on the server**

   The system needs to balance the work load and avoid frame time spikes on the server which can have a negative impact on the game experience.

5. **The system needs to be server authoritative**

   The system should have the authority on server side. Every propagation starts on the server.

6. **The propagation has to be deterministic**

   The propagation runs in a deterministic way. It avoids constant state replication across multiple clients sharing a game world. A propagation with identical parameters should result in an identical state on server and client without the need to replicate state updates every frame. Consequently, not relying on constant state replication makes the system less prone to lags.

4.2 Object-based Approach

Different from the system implemented for *Far Cry 2*, the main idea behind this approach is that fire will only jump from object to object. It relies on a high density of flammable objects in the game world.

4.2.1 Concept

The propagation in this concept is driven by two components:

- `FireComponent`
- `FireInstance`

A `FireComponent` represents flammable objects in the game world, but does not actively propagate fire. A `FireInstance` is not visually represented within the game world. It is a structure that propagates fire without burning itself, it is merely a container for updating `FireComponent` instances.

FireComponent

Every flammable object has a collection of fire-related properties that define how it burns and propagates.
These properties are member variables of the `FireComponent` class, which can be added to any entity in the game world. The member variables of the `FireComponent` class define when an entity transitions from the `Default` state to the `OnFire` state

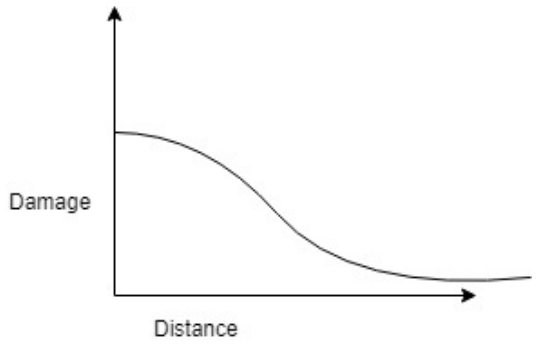

FIGURE 4.1: `damageDistanceScalingCurve` Graph

and finally to the `BurntDown` state. Both transitions are described by the following member variables:

- `health`
- `burningTime`

The values for these can be set per entity type.
`health` determines how fast an object will catch fire and thus transition from the `Default` to the `OnFire` state. It doesn't give details about the absolute amount of seconds. Within this speed `health` is decreasing. This finally triggers the state transition to `OnFire` when it reaches zero, depending on several factors that will be explained later in this chapter.
To simulate the behavior of an explosion leaving burnt objects, a specific property for that purpose was added. When set the flag `burnDownInstantly` triggers a state transition from `Default` over `OnFire` to `BurntDown` in one frame. All `FireComponent` instances are stored within cells of the `SpatialGrid` class. As a result each `FireComponent` instance has a three-dimensional `position` and a `radius`. In addition to `radius` the `FireComponent` class holds the member variable `propagationRadius`. It does not correspond with `radius`, but plays an important role for the propagation algorithm, which is explained in 4.2.1.
When `FireComponent.health` is decreased by a `FireInstance` instance (4.2.1) applying damage to it, the absolute amount of damage is scaled by `damageDistanceScalingCurve`. This member variable is defined as a two dimensional graph (4.1). The vertical axis is the absolute damage clamped between 0.1 and 1, the horizontal axis is the distance, which is clamped between 0 and the combined radius of the `FireComponent.radius` and `FireInstance.radius`. It determines the absolute amount of damage the `FireComponent` class receives, scaled by the distance between the `FireInstance.position` and the `FireComponent.position`. This concept, especially the combination of `health`, `burningTime` and `propagationRadius` enables a rich variety of burn and propagation behavior.

Examples A **tall tree** should propagate fire differently to a dry patch of grass does. A tree takes long to start burning, burns for a long time and spreads the fire to surrounding plants. To model this kind of behavior the values of the three mentioned member variables have to be set like the following for an entity that is representing a **tall tree**:

- High `health`
- High `burningTime`
- Big `propagationRadius`

Contrary to this is the modelling the burning and propagation behavior of a **dry patch of grass**. The following setup is be required:

- Low `health`
- Low `burningTime`
- Small `propagationRadius`

A **dry patch of grass** catches fire very fast and burns down fast. It only spreads the fire to plants in direct contact as it does not burn as hot as a tree.

FireInstance

Although the `FireComponent` class describes how entities propagate, it doesn't actively propagate fire itself. In this concept the `FireInstance` class is the active propagator. It actively decreases the `health` of `FireComponent` instances and thus indirectly triggers state transitions from `Default` to `OnFire` and finally to `BurntDown`. The `FireInstance` class is not a component and not connected to a specific entity. Its creation is triggered by events of different origins. For example an explosion or the impact of a fire arrow on a flammable material.
To define how the `Fire Instance` class affects the `FireComponent` class, four member variables have been introduced:

- `damageTimer`
- `damagePool`
- `radius`
- `isExplosion`

DamagePool represents the total amount of damage a `FireInstance` instance can deal to `FireComponent.health`. As the absolute amount of damage applied is depending on the distance between `FireInstance.position` and `FireComponent.position`, `damagePool` represents the maximum amount of damage n to a single `FireComponent` instance. For m `FireComponent` instances the

maximum amount of damage is $\frac{n}{m}$ per class instance. As the damage value in `FireComponent.damageDistanceScaleCurve` is clamped between 0.1 and 1, the minimum amount of damage for m `FireComponent` instances is $0.1 * \frac{n}{m}$ per instance. A high `damagePool` will increase the amount of propagation and potentially set more entities on fire.

DamageTimer defines the time span for a reduction of `damagePool` by n, for n `FireComponent` class instances. The reduction of `damagePool` by n is equivalent to a reduction of `FireComponent.health` by a value between $0.1 * n$ and n for n `FireComponent` class instances. Thus a low `damageTimer` will decrease the `damagePool` faster. The resut is a faster propagation.

Radius defines the area of effect a `FireInstance` instance initially has. `radius` is used to determine the affected `FireComponent` instances (4.2.1).

IsExplosion triggers state transitions for `FireComponent` instances within `radius` in one frame similar to `FireComponent.burnDownInstant`. Each `FireComponent` instance that receives damage when `isExplosion` is true will immediately transition to `BurntDown` regardless of the value of `FireComponent.health`. This action will reduce `damagePool` by n for n `FireComponent` instances.
In a game context it is possible to model different kinds of behavior for igniters. Two of the most common incendiary weapons in a video game are a fire arrow and a Molotov cocktail. Of these two examples the Molotov Cocktail burns longer and effects a larger area. This behavior can be modeled by assigning a high value to `damagePool` and a large value to `radius`. A fire arrow on the other hand requires a small value for `radius` and `damagePool`, since its area of effect is very limited.

Rules of Propagation

Although all `FireComponent` instances are stored in a `SpatialGrid` instance the propagation itself is not based on this grid structure. The rules of propagation have a high impact on how the player perceives it in the game world. As one of the requirements(1) the fire spreads in a believable way. Furthermore, the propagation itself needs to take other requirements into account, mainly the ones regarding performance.

Main Idea The first iteration of the concept includes creating a `FireInstance` class instance for each `FireComponent` class instance. Although providing an accurate and believable way of spreading, this does not scale well with big game worlds full of flammable entities, a forest for example. A hogh amount of `FireComponent` instance need to be created. For each one in the state `OnFire` a `FireInstance` class instance needs to exist. Additionally this has to be updated in every frame. Therefore the introduction of the concept of growing `FireInstance` instances was preferred.

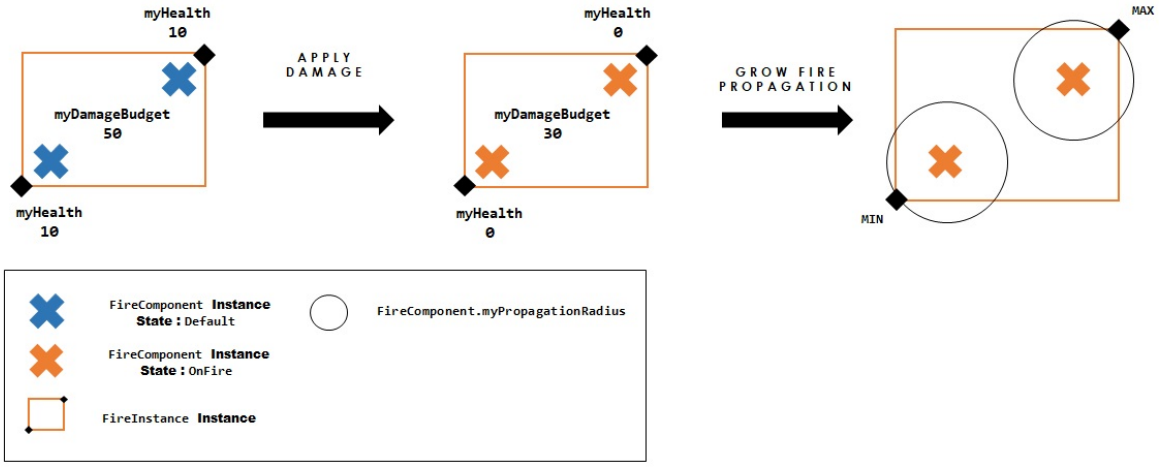

FIGURE 4.2: `FireInstance` growing
A `FireInstance` instance with a single element in `FirePropagationAABBs` is damaging two `FireComponent` instances. When both their `health` values reach zero, the element in `FirePropagationAABBs` grows.

Instead of creating a `FireInstance` instance per burning entity, its area of effect increases over time. It is represented by `FireInstance.PropagationAABBs` where each element defines a sphere. The abbreviation AABB stands for axis-aligned bounding box. Although technically using a sphere for calculations, the area of effect is stored as AABB. This is due to implementation details of the `SpatialGrid` class. Each `FirePropagationAABB` instance additionally stores the spatial hash of the cell it the center of it would be placed in. It is not stored within a`SpatialGrid` instance.

FirePropagationAABB Creation The position and radius of the sphere used to create the first element of `FireInstance.PropagationAABBs` is not defined within the system. While the radius is set manually, the position is set by the event that triggers the creation of a `FireInstance` instance. For example the impact of a fire arrow on flammable material, which uses the impact position as initial position of a `FirePropagationAABB` instance. A `FireInstance` instance will start reducing the value of `health` for each `FireComponent` instance where the sphere defined by `FireComponent.radius` and `FireComponent.position` intersects with the AABB defined by a `FirePropagationAABB` instance. No other factors are taken into account for this step.

Damage and Growing The member variable `damageDistanceScalingCurve` is added to give the reduction of `FireComponent.health` a falloff resulting in a smoother propagation. An important thing to mention is, that the visual representation of the fire is not part of the system in this approach.
Once the value of `health` of a `FireComponent` instance reaches zero, it transitions

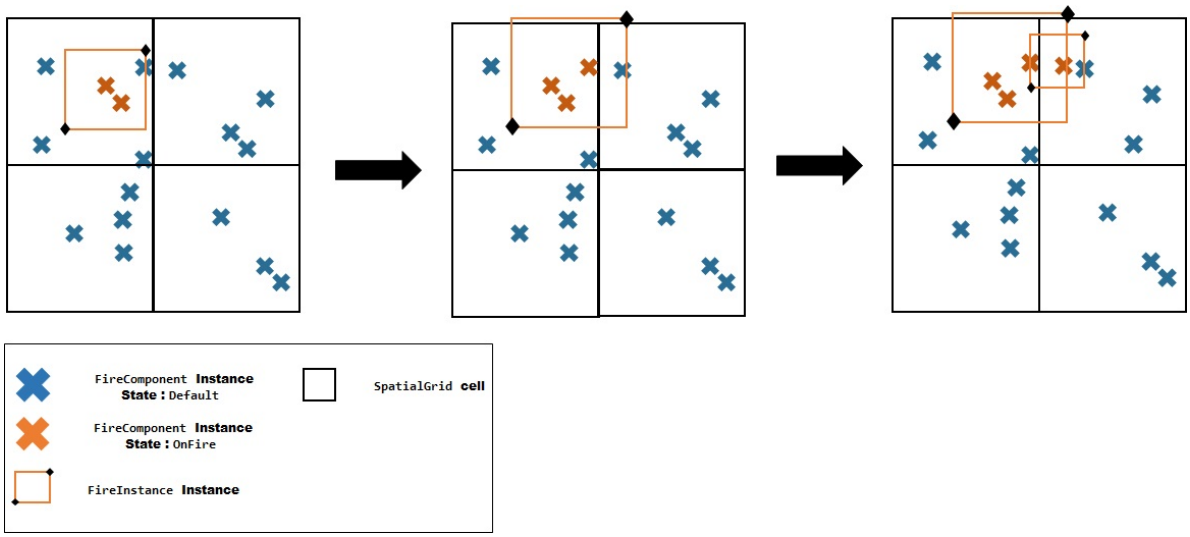

FIGURE 4.3: Fire Propagation

to the `OnFire` state. When this transition occurs, the `FireInstance` instance triggering this state transition grows. This is happening either by growing an existing element in `FireInstance.PropagationAABBs` or adding a new one to the array. To decide, whether to add a new one or grow an existing one, the spatial hash is needed. If an element in `FireInstance.PropagationAABBs` shares the same spatial hash with the `FireComponent` instance set on fire, this element will grow its bounds. Before growing an AABB out of `FireComponent.position` and `FireComponent.propagationRadius` has to be constructed. This constructed AABB is merged with the element in `FireInstance.PropagationAABBs`, resulting in an AABB containing both initial AABBs, which is then reassigned to the element in `FireInstance.PropagationAABBs`.
If the spatial hash of `FireComponent.position` does not match with any element in `FireInstance.PropagationAABBs`. The constructed axis-aligned bounding box will be assigned to a new element that is then added to `FireInstance.PropagationAABBs`.
This process is visualized in Figure 4.3. The process of adding or growing a `FirePropagationAABB` is triggered by every state transition of a `FireComponent` instance from `Default` to `OnFire`, as long as the value of `FireInstance.damagePool` is not zero or smaller.

Lifetime and Shrinking The lifetime of a `FireInstance` instance depends on several factors. It is considered alive as long as the value of `damagePool` hasn't reached zero and at least one `FireComponent` instance is within an element of `FireInstance.PropagationAABBs`. Its lifetime also heavily depends on whether a `FireComponent` instance is within or intersecting with the first element of `FireInstance.PropagationAABBs`. If this is not the case, the `FireInstance` instance will be destroyed immediately.

Elements within `FireInstance.PropagationAABBs` do not necessarily exist over the whole lifetime of a `FireInstance` instance. Multiple conditions have to be met for a `FirePropagationAABB` instance to be destroyed. These conditions are different for the first element in `FireInstance.PropagationAABBs` than for all other elements. For the first element the conditions that have to be met are either of following:

- All `FireComponent` instances within or intersecting are in the state `BurntDown`.
- There are no `FireComponent` instances within or intersecting.

For all other elements the following condition has to be met:

- No `FireComponent` instances within or intersecting are in the state `OnFire`.

This difference is due to the fact that the creation of a new `FirePropagationAABB` instance is triggered by a state transition of a `FireComponent` class instance from `Default` to `OnFire`. It ensures att least one `FireComponent` instance is in the state `OnFire`. This is not the case for the first element in `FireInstance.PropagationAABBs`, which can be created at any position and does not necessarily have any `FireComponent` class instances within or intersecting with its axis-aligned bounding box.
By adding this functionality, the `FireInstace` class can grow or shrink its area of effect. While this is true for the combined area of effect of all elements in `FireInstance.PropagationAABBs`, the `FirePropagationAABB` class can only grow its AABB while it doesn't have the functionality to shrink it.

Ownership There is no restriction to how many `FireInstance` instances can exists at the same time or at the same position. A `FireComponent` instance can be within multiple elements of `FireInstance.PropagationAABBs` of multiple `FireInstance` instances. While the `FireComponent` instance is still in the state `Default`, it can be damaged by multiple `FireInstance` instances. Once the value of `FireComponent.health` reaches zero, the `FireInstance` instance applying damage last will store a pointer to this `FireComponent` instance. This enables keeping track of all `FireComponent` instances a `FireInstance` class instance set on fire during its lifetime.

Multiple Fire Instances With the presented concept it is possible to have overlapping areas of effect of multiple `FireInstance` class instances. The number of `FireInstance` class instances damaging a `FireComponent` class instance simultaneously is not limited, but only one can trigger the state change to `OnFire`. When this happens, the `FireInstance` instance triggering the state change will reference the `FireComponent` instance.

Synchronization The described system is intended to simulate the Fire Propagation on both, client and server simultaneously. Since the system is deterministic, the results of both simulations will match. To make sure both simulations run in sync and the simulation itself has not been modified on the client, regular state updates have to be sent from server to client. This system is server authoritative, the server dictates the current state of the propagation and sends it out to all connected players. This means no events or status updates related to the Fire Propagation System will be sent from client to Server.

4.2.2 Implementation

As *Snowdrop* was written in *C++*, the implementation, although not showing *C++*-code, uses concepts of the *C++*-language. This thesis only presents the structure of the implementation and provides pseudo-code for all important functions.

Structure

This system was implemented for an MMOG with an dedicated server-client structure. Because of this, many classes have a shared base which client side and server side classes are derived from. These derived classes typically override virtual functions or add new functionality. The implementation thus introduces the prefixes `Shared_`, `Client_` and `Server_`. Which respectively stands for the base class, the client side derived class and the server side derived class. If no prefix is given, it is always the base class that is being referred to. The shared base class contains most of the functionality, as it was one of the requirements to have a deterministic simulation running on both, client and server-side. `FireComponentManager` and `FireManager` are the two central classes in this implementation. Both classes are part of the `WorldModel` class, because it has a pointer to each as member. It is important to note, that both derived classes of the `WorldModel` class create derived class instances of the `FireComponentManager` and the `FireManager` class, but they are both assigned to pointers to their base classes, which are members of `Shared_WorldModel`.
The `FireComponentManager` class handles the creation and destruction of `FireComponent` instances, as well as inserting them in `SpatialGrid`, which it stores in one if its members. The `FireManager` class on the other hand holds all `FireInstances` and a reference to a `FireComponentManager` instance to access the `SpatialGrid`. For performance and memory reasons, all members of the `FireComponent` class were split into constant and non-constant data. All non-constant data was added to the `FireState` class. This class holds information about the current state of a `FireComponent` instance in each `WorldModel` instance. The constant data is only stored in the `WorldModel` class instance that represents the *Global World Model*. The `FireState` class has a member that points to a `FireComponent` instance holding the constant data for the same entity. An overview of all classes in the implementation can be seen in figure 4.4.

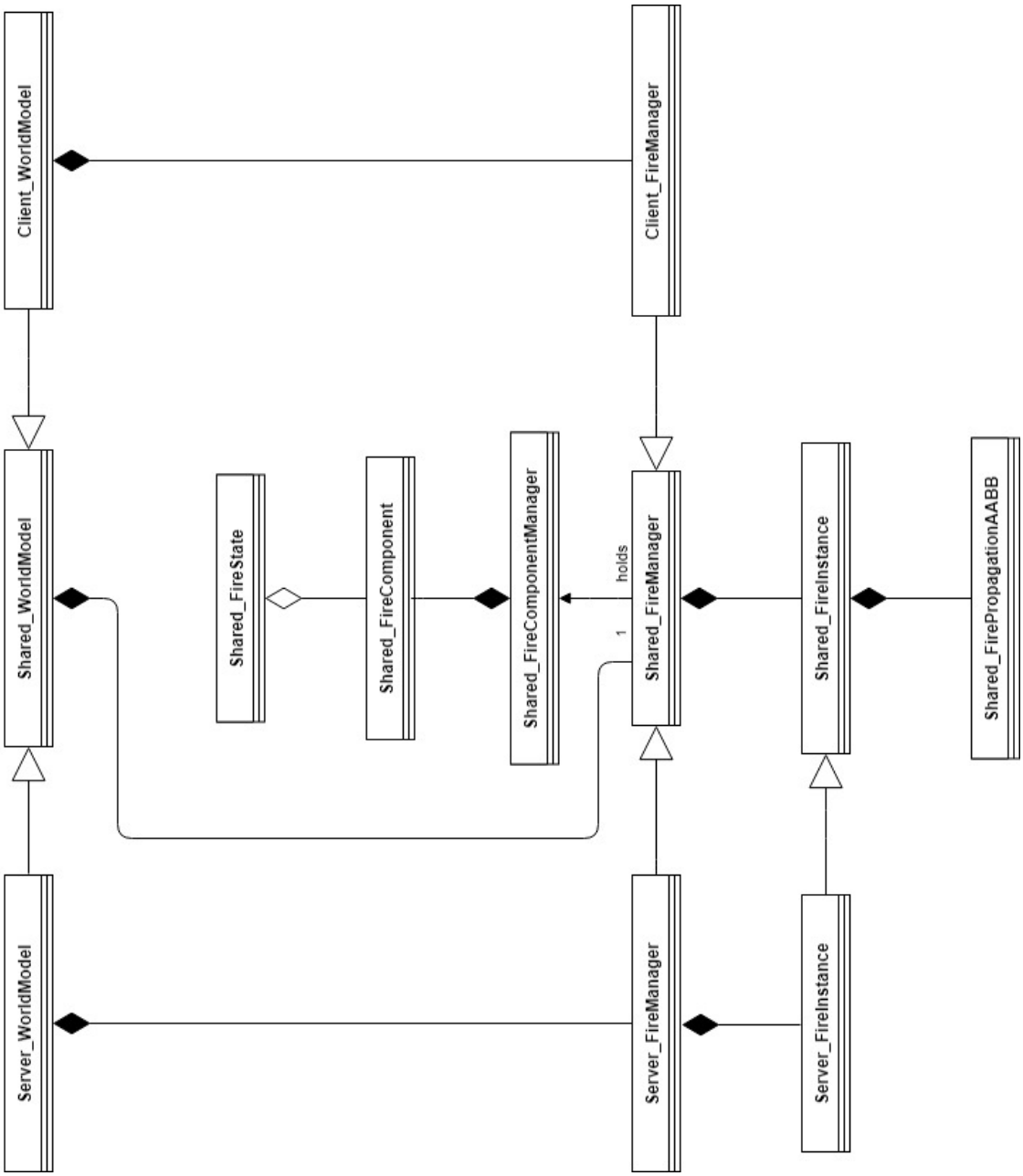

FIGURE 4.4: Implementation Overview

Network Overview

Although being a deterministic simulation, the Fire Propagation System is required to work with an authoritative dedicated server. This section provides a short overview of how the dedicated server communicates with all connected clients in the context of this implementation.
Of the three kinds of replication listed in 2.1.2, only events were used. Events are created and sent to the client by the `Server_NetworkOutputManager` class. This class is not derived from a shared base class and only exists on the server. It implements a function for each group of events associated with a system and retrieves data, which is assigned to a newly created event, from it. Incoming events are handled by the `Client_InGameNetworkEventHandler` class, which then calls the function associated with the incoming event. This association is defined within the `Client_InGameNetworkEventHandler` class and is not part of this implementation.

Network Events As the system is deterministic, almost no *State Data* or *Control Data* is needed. The synchronization happens via *Events* and regular transmission of *State Data*, which only includes a few selected properties. As the server is authoritative, events are usually sent from server to client, in some rare cases from client to server. In this implementation they only contain an array of a single class as member variable. This data is being serialized before the event is send. For this, each event implements two functions, `Init()` and `Process()`. An event is not initialized with data, the data is a parameter of the function `Init()` in which it is copied to the member variable, an array of the same type. In the `Process()` function the copied data is serialized.
To keep client and server side of the system in sync, two network events were created. The `FireManagerSpawnNetworkEvent` class communicates every created `Fire Instance` instance from server to client, while the `FireManagerUpdateNetworkEvent` class contains state transitions from `Default` to `OnFire` and from `OnFire` to `BurntDown`. The `FireInstanceCreationParams` class contains the data for the `FireManagerSpawnNetworkEvent` class. Its members allow replication of a `FireInstance` instance.
The `FireManagerUpdateNetworkEvent` class on the other hand, contains an array of `FireComponentStateChange` instances. Its member variables define the unique identifier (UID) of an entity, and what state it transitioned to.
The function `BuildFireManagerOutput()` checks first whether any data for an event is present by calling `HasCreatedFireInstaces()` and `HasComponentStateChanges()`. If data is present, `FillFireInstancesCreationParams()` and `FillFireComponentStateChanges()` copy this data to the member variable of a newly created `FireManagerUpdateNetworkEvent` or `FireManagerSpawnNetworkEvent` instance.

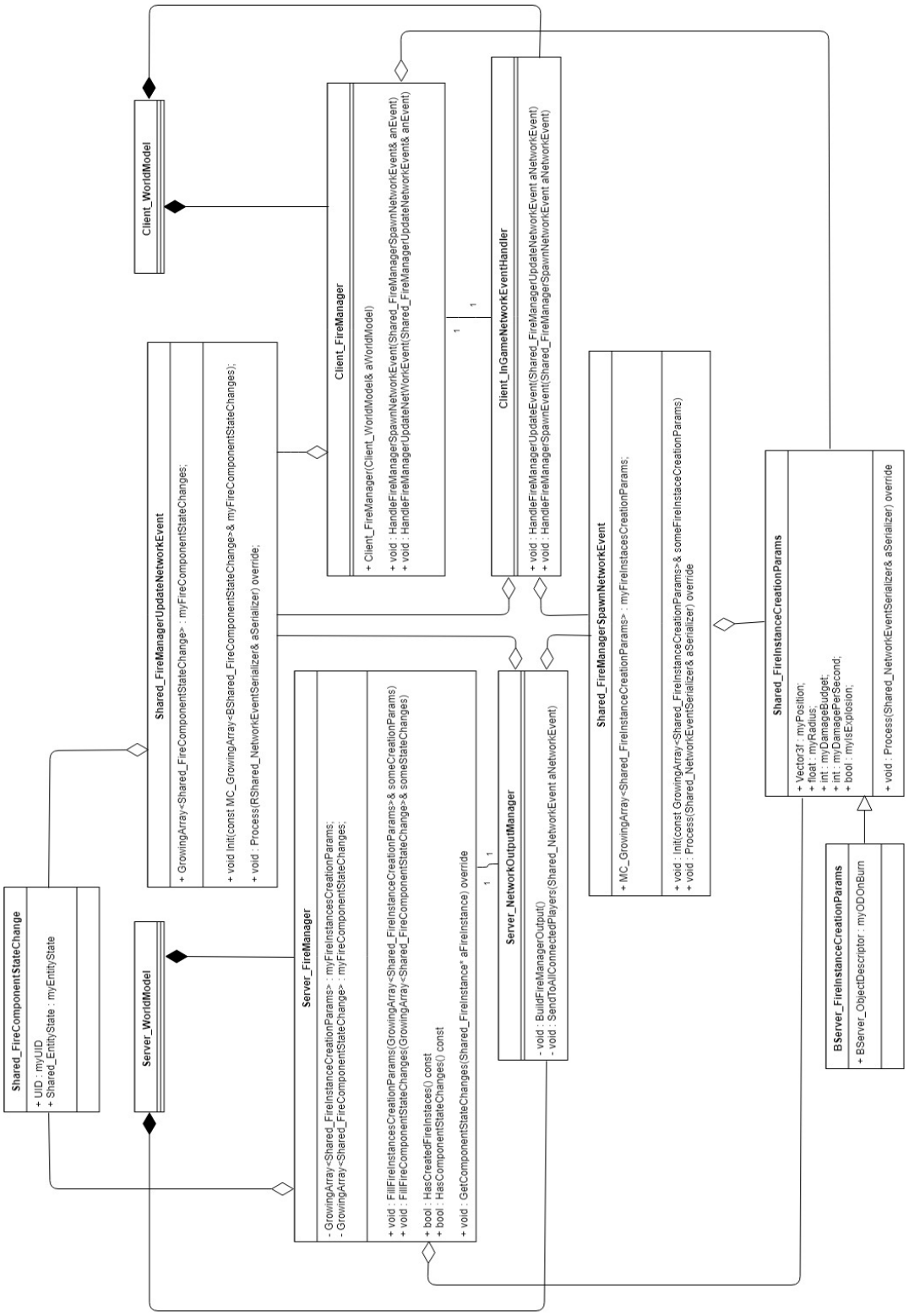

FIGURE 4.5: Network Overview

The `Client_InGameNetworkEventHandler` class implements functions that handle the incoming events:

- `HandleFireManagerUpdateEvent()`
- `HandleFireManagerSpawnEvent()`

These functions pass on the event to the `Client_FireManager` class, which then loops over the event data and either creates new `FireInstances` instance or triggers a state transition for `FireComponent` instances.

FireComponent

The `FireComponent` class was created with the Entity-Component-System Pattern 3.1 in mind. It only holds constant data that is used by the Fire Propagation System. All concept-relevant members were already discussed in 4.2.1. This section will list implementation specific members.
The prefix `my` is part of the naming convention for this implementation and is used for all member variables. For static members, the prefix `our` is used.
In general, all variable names start with a lower case letter, while class names and function names start with an upper case letter. An exception to this rule are values of an enumeration.
All `FireComponent` instances are stored in an intrusive linked list, which is part of the `FireComponentManager`. This data structure stores the list node as a member in each entry, which is represented by `myLink`.
`myEntityID` represents the UID of the entity each instance is attached to. It is stored as a 32-bit integer, which is part of the `EntityID` class.
The data type `myDamageDistanceScalingCurve`, Curve, is a class that stores key-value pairs and offers multiple ways to interpolate between those values.
`FireComponent` class instances are owned by `FireComponentManager`.

FireState and PhaseStateComponent The `FireState` class is a member of the `PhaseStateComponent` class which holds all non-constant data for all attached components of a single entity. The `FireState` class holds all dynamic data specifically for the Fire Propagation System, it keeps a pointer to a `FireComponent` instance that contains all constant data.
`myHealth` and `myBurningTime` are two member variables that change their values as soon as a `FireState` instance is affected by a `FireInstance` instance. `BurnDownInstantly`, a variable used in the concept, was split into two members, `mySetOnFireNextDamageReceived`, `myExtinguishNextDamageReceived`.
The function `ApplyDistanceScaledDamage()`(4) is responsible for calculating damage to `myHealth`. When the value of `myHealth` reaches zero, the internal state of the `FireState` instance is equivalent to the entity state `OnFire`, although it is set separately. Both, the entity state of an entity and the internal state of a `FireState` whose

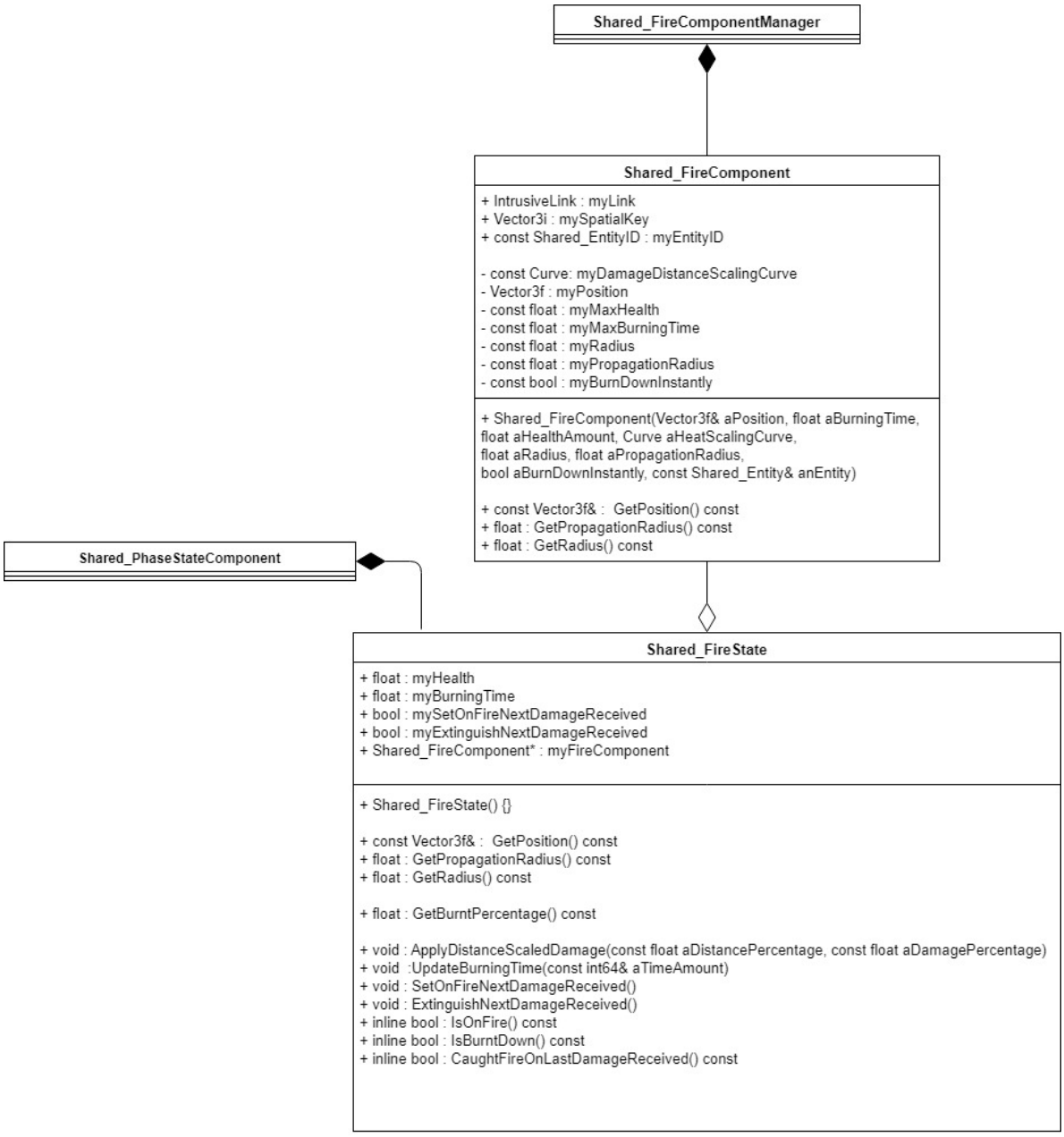

FIGURE 4.6: `FireComponent` UML

`myFireComponent` member points to a `FireComponent` instance attached to this entity must be in a consistent state. The internal states of the `FireState` class are all equivalent to an entity state and are defined as followed:

- `Default` :
  When the value of `myHealth` and `myBurngingTime` is bigger than zero, the `FireState` instance is in the equivalent of the `Default` state.
- `OnFire` :
  When the value of `myHealth` is zero but `myBurningTime` is not, the `FireState` instance is in the equivalent of the `OnFire` state.
- `BurntDown` :
  When the value of `myHealth` and `myBurngingTime` is zero, the `FireState` instance is considered is in the equivalent of the `BurntDown` state.

Function 1: `FireState::ApplyDistanceScaledDamage()`

Input: *aDistance*, *aDamage*; `with value v` $0 <= v <= 1$
Output: `No output`
if (*IsBurntDown*() *returns true*)
{
    `return`
}

myHealth = *myHealt* − *aDamage*∗ `y-value of` *damageDistanceScalingCurve* `at x` = *aDistance*

if (*mySetOnFireNextDamageReceived is true*)
{
    myHealth = 0
    setOnFireNextDamageReceived = *false*
}

When in the internal state `OnFire`, instead of `ApplyDistanceScaledDamage()`, the function `UpdateBurningTime()`(2 is called to decrease the value of `myBurningTime` until it reaches zero and the internal state transitions to `BurntDown`.

Function 2: `FireState::UpdateBurningTime()`

Input: *aTimeAmount*
Output: `No output`

burningTime = *burningTime* − *aTimeAmount*
if (*myExtinguishNextDamageReceived is true*)
{
    myBurningTime = 0 *myExtinguishNextDamageReceived* = *false*
}

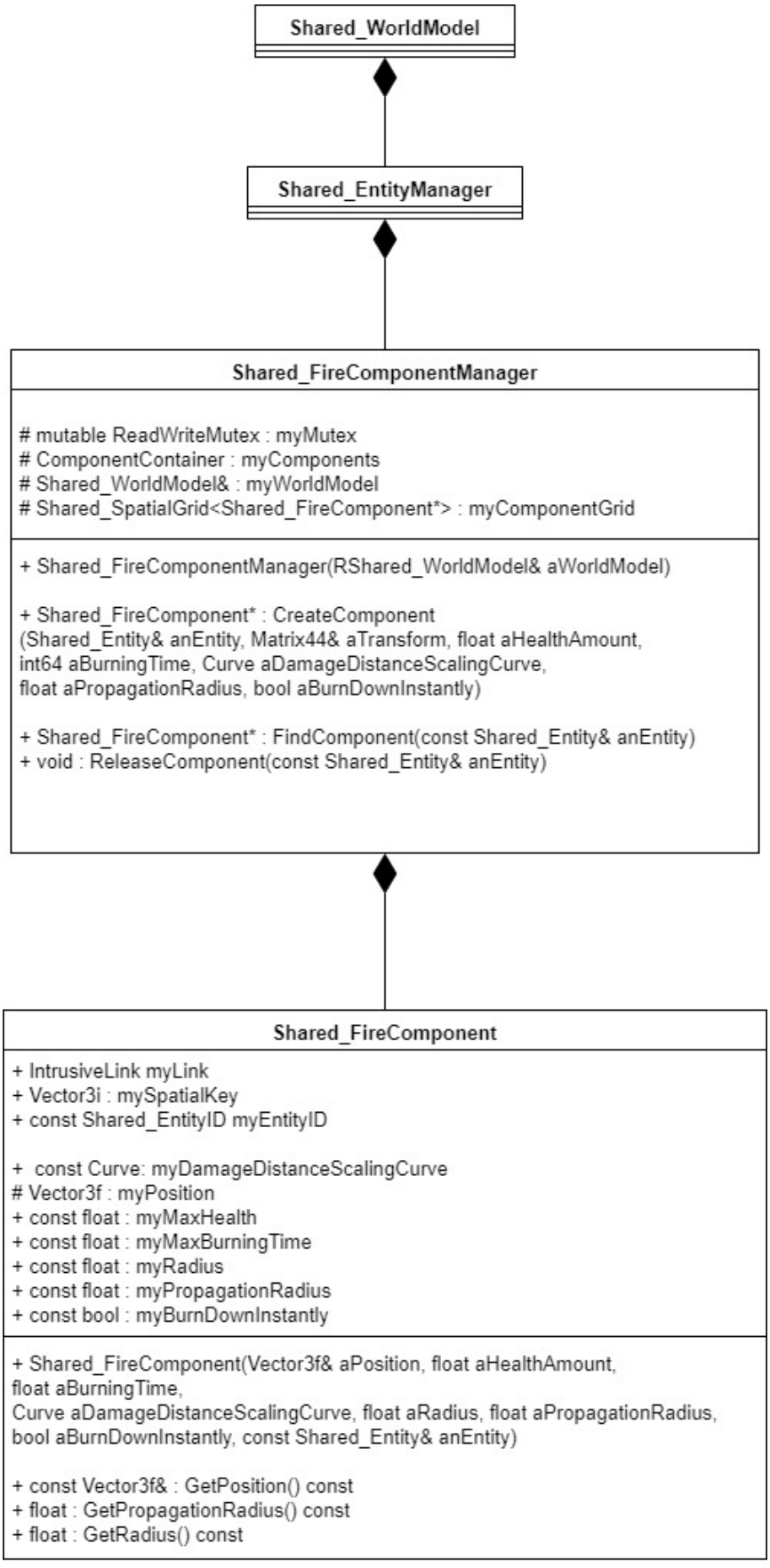

FIGURE 4.7: `FireComponentManager` UML

FireComponentManager

The `WorldModel` class contains a pointer to a `FireComponentManager` instance. It stores, creates and manages the life time of all `FireComponent` instances. After creation, each `FireComponent` instance is inserted into an intrusive list, implemented by `ComponentContainer` and into a spatial hash grid, implemented by `SpatialGrid`. There is a slight distinction between the `FireComponentManager` instance in the *Global World Model* (3.3) and instances in the *Server-* and *Client World Models*. As listed in 4.2.1, all members of the `FireComponent` class are either marked as `const` or not intended to change during the life cycle each `FireComponent` instance. Since this constant data would be the same for all players connected to same dedicated server, there is no need to create an instance of the `FireComponentManager` class for each existing Server and Client World Model. Instead, `FireComponentManager` instances created in *Server-* and *Client World Models* store a pointer (`myParent`) to the existing `FireComponentManager` instances of the *Global World Model*.
The *Global World Model* is created on startup for the dedicated server process, the *Server-* and *Client World Models* only when players connect to the dedicated server. This means all entities and their components already exist within the *Global World Model* when a player connects to a dedicated server. An exception to this behavior is the `PhaseStateComponent` class, which is created per entity in every *Client-* and *Server World Model*. The `FireComponentManager` class itself does not know in which *World Model* it was created. If `$myParent != nullptr$`, a *Client-* or *Server World Model* is assumed. Is this the case, a `FireComponent` instance that contains all the constant data already exists in the *Global World Model*. As all entities are replicated for each player within a game world, they share the same constant data. The `FireState` class can retrieve a pointer to an existing `FireComponent` and store it within `FireState.myFireComponent`, this happens within the `CreateComponent()` function.
The `FindComponent()` function simply retrieves a pointer to a `FireComponent` instance if it exists either within the `FireComponentManager` pointed to by `myParent` or the `FireComponentManager` instance the function is called on. If no `FireComponent` instance is found for the entity passed in as parameter, `nullptr` is returned. As described in 3.2, all systems are assumed to run in a multi-threaded environment. For this purpose, `myMutex` was added as a member variable. It is locked before each access to `myComponents` or `myComponentGrid`. As explained in 3.4, the game world is split into sectors. This was introduced to save memory by only loading sectors and all contained entities around the player. For that reason, components are created and released frequently throughout every play-session.

FireManager

The `FireManager` class handles creation and deletion of all `FireInstance` instances. It is responsible for collecting event data and handling incoming events

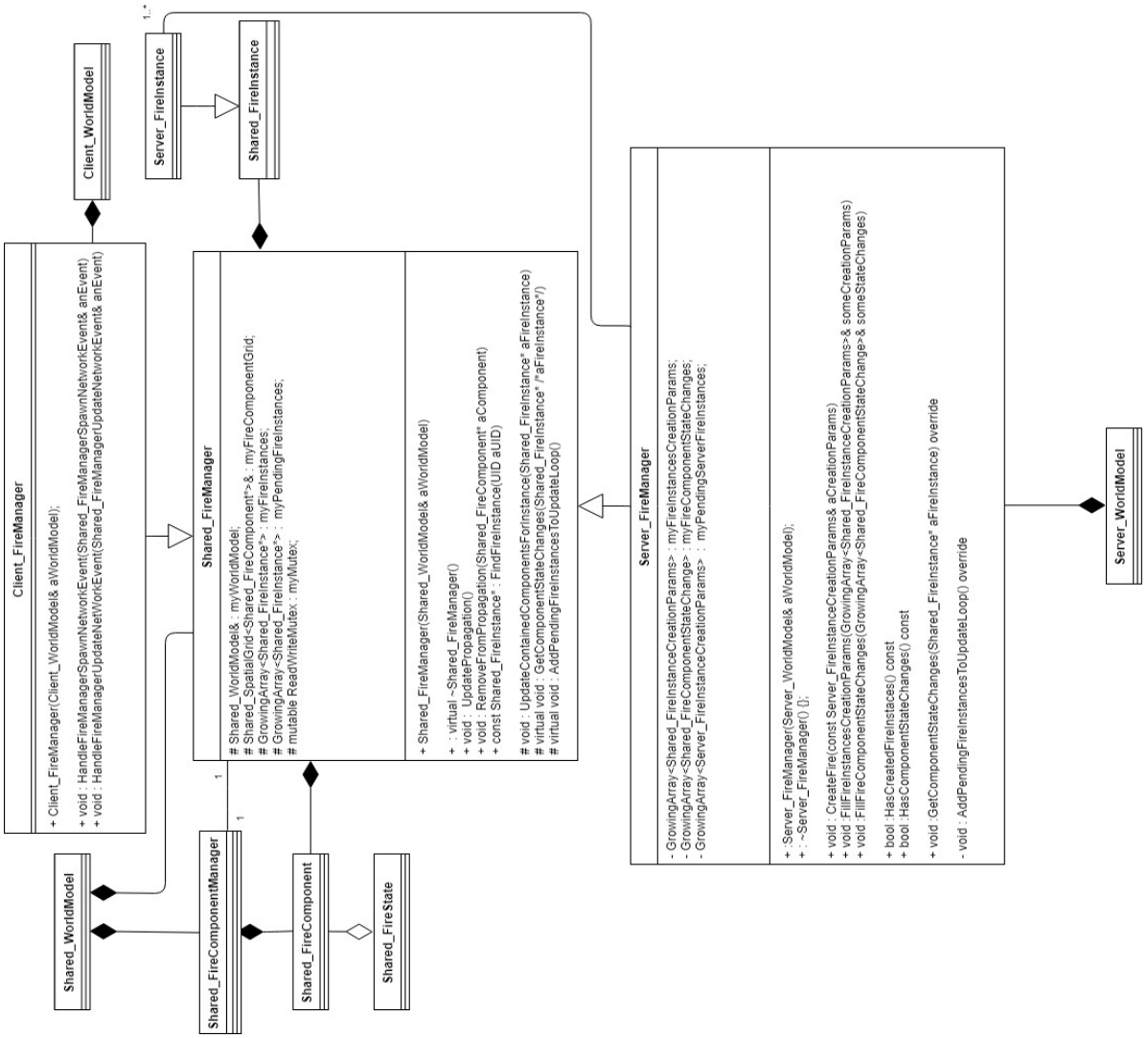

FIGURE 4.8: `FireManager` UML

on client side. The `myMutex` member is locked whenever `myFireInstances` is modified. By having a separate array `myPendingFireInstances`, I prevented having to lock `myMutex` for the entire `UpdatePropagation()` function. Each new `FireInstance` instance is added to `myPendingFireInstances` first, before the `AddPendingFireInstancesToUpdateLoop()` function copies all entries to `myFireInstances`. Instead of creating accessor functions, I decided to store a pointer to `FireComponentManager.myComponentGrid` in `myFireComponentGrid`. As the `FireManager` class is declared a `friend` class to the `myFireComponentGrid`, it can access this member even though it is `private`.
Furthermore, the `FireManager` class stores a pointer to a `WorldModel` instance. This needs to be accessed as soon as other systems or components of a different type are needed, for example the `EntityStateComponent`. The `UpdatePropagation()`(3) function is called from the `World Model` class, which happens on client and server side. Within this function, the current state of the propagation is updated, damage to `FireState` instances is applied, entity state changes are triggered and all `FirePropagationAABB` instances are updated.

Function 3: `FireManager::UpdatePropagation()`

```
Input: No input
Output: No output
for (each fireInstance in fireInstances)
{
    fireInstance.ApplyDamage()
    GetComponentStateChanges(fireInstance)
    if (fireInstance has grown its bounds)
    {
        UpdateContainedComponents(fireInstance)
    }
    if (fireInstance did not grow and is not on fire)
    {
        WriteLock(mutex)
        fireInstances.Delete(fireInstance)
    }
}

fireInstances.Add(pendingFireInstances)
```

The `RemoveFromPropagation()` function is exclusively called from the `FireComponentManager` class when an entity is unloaded. This happens either when a sector is unloaded, or when the the player quits the game. It removes all references to the `FireState` instance for this entity from each element in `myFireInstances`.
The `UpdateContainedFireStates()` function is called when a `FireInstance` instance is created to determine the `FireState` instances that are within the first element in `FireInstance.myPropagationAABBs`. Furthermore, it is called when a `FirePropagationAABB` instance grows its bounds or is newly created and added to `FireInstance.myPropagationAABBs`. It is also responsible for removing `FirePropagationAABB` instances if the criteria, as explained in 4.2.1, are met.

Unlike the `FireComponentManager` class, the `FireManager` class has two derived classes, the `Server_FireManager` and the `Client_FireManager` class. Both implement virtual functions of their base class and contain additional functions.

Server_FireManager The `Server_FireManager` class overrides and implements the `AddPendingFireInstancesToUpdateLoop()` and `GetComponentStateChanges()` functions which are defined as `virtual` in the base class. This is due to the`Server_FireInstanceCreationParams` class being used to created `Server_FireInstance` instances. This only happens server side. Since the system is supposed to work in a server authoritative environment, gameplay effects like damaging non-player-characters (NPCs) or the player character have to be executed in the dedicated server process. For this reason, all fire damage is handled in `Server_FireInstances` which override certain virtual functions of their base class.
For damage applied by this system, existing system within *Snowdrop* were used. These systems are server driven without derived client classes. For this implementation, they will not be further discussed
The `Server_NetworkOutputHandler` class checks by calling the `HasCreatedFireInstances()` and `HasComponentStateChanges()` function whether any events should be replicated for all connected clients sharing the same game world. If this is the case a new `FireManagerSpawnNetworkEvent` or `FireManagerUpdateNetworkEvent` instance is created and filled with data by calling `FillFireInstacesCreationParams()` and `FillFireComponentStateChanges()`.
Calling `GetComponentStateChanges()` will retrieve all state changes of a `Shared_FireInstance` instance that were triggered since the last time damage was applied to it.

Client_FireManager On the client side, the `Client_FireManager` handles incoming events. The functions handling these events are exclusively called by the `Client_InGameNetworkHandler` class. Their purpose is to create `FireInstance` instances or trigger state changes that happened on the server, but not on the client yet, to keep both simulations in sync.

Fire Instance

The `FireInstance` class is responsible for dealing damage to `FireState` instances and thus initiating state transitions internally as well as for the `EntityStateComponent` class. Essentially, it stores an array of `FirePropagationAABB` instances which are used for search queries within the spatial hash grid of the `FireComponentManager`. All found `FireState` instances are stored as pointers. When a `FirePropagationAABB` instance grows, or a new one is created, this step is repeated. Pointers to `FireState` instances are stored in four different arrays, each one with a different purpose:

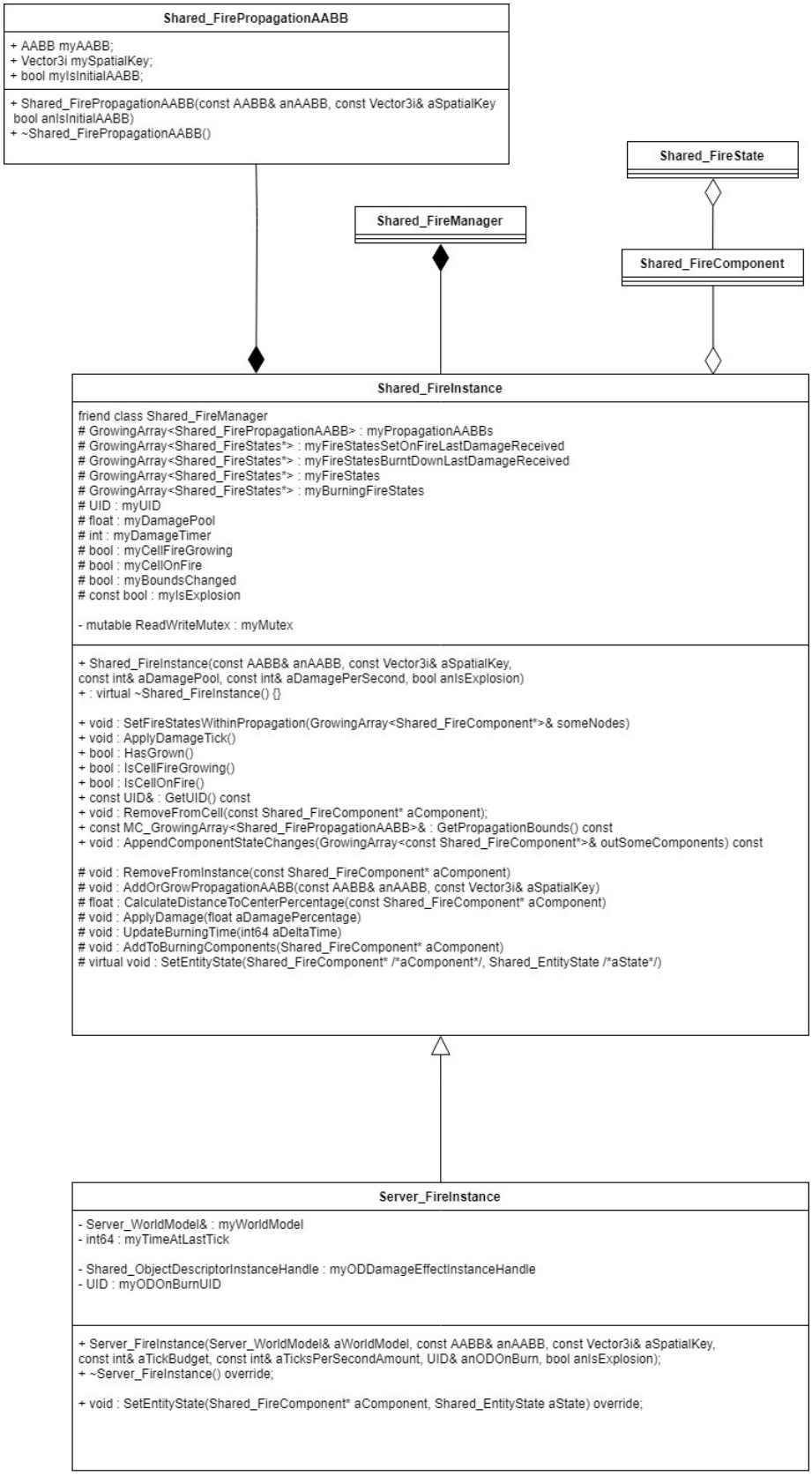

FIGURE 4.9: `FireInstance` UML

- `myFireStates`
- `myBurningFireStates`
- `myFireStatesSetOnFireLastDamageReceived`
- `myFireStatesBurntDownLastDamageReceived`

The function `SetFireStatesWithinPropagation()` is called by the `FireManager` class and copies all elements of the array passed in as an argument over to `myFireStates`. All elements in `myFireStates` will receive damage and are required to be in the `Default` state.
The `ApplyDamage()` functions loops over this array and removes all `FireState` instances that are not in the `Default` state. Next, if *myIsEplosion* == *true* the `SetOnFireNextDamageReceived()` function will be called on the current element in the loop, which will trigger a state transition to the `OnFire` state next time the `ApplyDistanceScaledDamage()` function is called on this `FireState` instance. This will happen during the same `ApplyDamage()` function call. Lastly, the `AddToBurningStates()`(5) function is called where a pointer to a `FireState` instance is passed in as argument when it is in the `OnFire` state.
In the `AddToBurningStates()` function, the passed pointer to a `FireState` instance will be added to `myBurngingFirestates` and `myFireStatesSetOnFireLastDamageReceived`. This function also transitions the state of the `EntityStateComponent` instance that is attached to the same entity as the `FireComponent` the `FireState.myFireComponent` member points to, to `OnFire`. Also, if the `myIsExplosion` member of this `FireInstance` instance equals `false`, it will grow or add an element in `FireInstance.myFirePropagationAABBs`. It is important to note, that the `AddToBurningStates()` function does not remove a `FireState` instance from the `myFireStates` member, this happens in the `ApplyDamage()` function.
The `UpdateBurningTime()` function has a similar structure as `ApplyDamage()`. It essentially updates the `FireState.myBurningTime` member of each `FireState` instance and triggers state transitions from `OnFire` to `BurntDown` as soon as the value of this member hits zero.
As damage is dealt per frame, the absolute amount has to be multiplied with the frame time of the previous frame. Additionally, it is multiplied with `myDamageTimer`. The `Server_FireInstance` class additionally handles the communication with the system damaging NPCs and player characters.
The `UpdateFireStates()` function is called from the `UpdatePropagation()` function of the `FireManager` class. It calls both, the `ApplyDamage()`(4) and the `UpdateBurningTime()` function. To ensure a frame time independent damage amount that is dealt based on the value of `myDamageTimer`, the frame time of the last frame is divide by `myDamageTimer`. The resulting value is passed to `ApplyDamage()` as an argument, while simply the frame time is passed to `UpdateBurningTime()`.

Function 4: `FireInstance::ApplyDamage()`

Input: *damageTimerPercentage*; `with value v` $0 <= v <= 1$
Output: `No output`
fireStatesSetOnFireLastDamageReceived`.Clear()`
for (*each fireState in fireStates*)
{
    if (*damagePool is zero or less*)
    {
        `return`
    }
    if (*fireState is OnFire or BurntDown*)
    {
        `WriteLock(`*mutex*`)`
        fireStates`.Remove(`*fireState*`)`
        `continue`
    }
    if (*isExplosion is true*)
    {
        fireState.setOnFireNextDamageReceived $=$ *true*
    }
    `CalculateDistanceToCenterPercentage(`*fireState.myFireComponent*`)`
    if (*fireState.CaughtFireOnLastDamageReceived()* == *true*)
    {
        burningFireStates`.Add(`*fireState*`)`
    }
    damagePool $=$ *damagePool* $-$ *damageTimerPercentage*
}

Function 5: `FireInstance::AddToBurningFireStates()`

Input: *aFireState*
Output: `No output`
if (*aFireState already exists in burningFireStates*)
{
    `return`
}
`WriteLock(`*mutex*`)`
burningFireStates`.Add(`*fireState*`)`
if (*isExplosion is false*)
{
    `AddOrGrowPropagationAABB(`*aFireState*`.`*fireComponent*`.`*mySpatialKey*`,AABB of` *fireState* `position and radius`
}
`set the state of the entity connected to this` *fireState* `to` *EntityState.OnFire*
fireStatesSetOnFireLastDamageReceived`.Add(`*fireState*`)`

4.2.3 Limitations

The functionality the presented implementation provides is very basic. Several identified cases are not handled by the system in its current state.

- The system does not check if there are barriers between flammable objects.
- If the player disconnects from the server during an active fire, the state of the fire will not persist.
- The system is not familiar with the concept of water

Furthermore, the concept limits the scale of usage for this system. In this concept, every flammable objects needs to be updated constantly as the visual aspect is handled for each object individually. This can result in a major performance impact depending on the size of the world and the number of `FireComponent` instances in it.

4.3 Cell-based Approach

The main idea behind this approach is that fire will propagate on the terrain itself and is not limited to flammable objects.

4.3.1 Concept

This concept is loosely based on the basics of Fire Dynamics explained in 2.2. Instead of using abstract terms like health and damage, which are very commonly found in games, this approach is based on the physical concept of temperature and heat transfer. Each Law of Thermodynamics was evaluated in terms of feasibility to translate it into game context. In the context of a game, the first law of thermodynamics could be translated as follows. Considering the game world as an isolated system, according to the law, its total energy is constant and thus can not be created or destroyed, but only transformed to another form. As the game world of the project did not feature any comparable feature, the implementation of a complete system based on the first law of thermodynamics was deemed too complex with too little impact on the gameplay. With the formula 4.1, the second and the zeroth law of thermodynamics are incorporated into the concept. The third law of thermodynamics on the other hand was not considered for the developed approach, because it was not clear how defining a minimum possible energy at absolute zero would benefit the gameplay.

Heat Transfer

After evaluating the concepts of conduction, convection and radiation, a simplified combination of all three was developed. The amount of thermal energy transferred

from one object to another is computed by the following formula:

$$\frac{\Delta Q}{\Delta t} = k * \Delta T \tag{4.1}$$

where:

$\frac{\Delta Q}{\Delta t}$ is the thermal energy ΔQ transferred per unit time Δt
k is the thermal conductivity of the material/object
ΔT is the temperature difference between two objects.

This equation allows control over the thermal energy ΔQ by adjusting the value of k, while providing a more interesting behavior than a linear temperature increase. It incorporates the Second and the Zeroth Law of Thermodynamics:

- If two objects have the same temperature as a third object, no thermal energy will flow between them
- Thermal energy can never flow from a colder to a hotter object

The output of this equation is simply added to the temperature of any target object. This approach differentiates between two types of object within the concept of Heat Transfer:

- a *Heat Source*
- a *Heat Receiver*

While the first actively transfers thermal energy to other objects, the Heat Receiver is only receiving thermal energy and does not transfer it further.

FireBurnSettings

The `FireBurnSettings` class consist of a group of variables that are used by the Fire Propagation System. It provides a unified structure that describe the behavior of both, *Heat Sources* and *Heat Receivers*. All members of this class are constant and do not change during the propagation:

- `temperatureThreshold`
- `thermalConductivity`
- `initialBurningTemperature`
- `burningTemperatureChangePerSecond`
- `propagationTime`

These parameters were chosen with the Fire Development curve 2.2.3 and the equation 4.1 in mind. Every flammable entity holds a variable that describes its current temperature. If this temperature rises above the value of `temperatureThreshold`, it transitions from the `Default` state to the `OnFire` state. The `thermalConductivity` represents `k` in the equation 4.1 and modifies the thermal energy that is transfered and received. To allow more control over the temperature development while an object is burning, `initialBurningTemperature`, `burningTemperatureChangePerSecond` and `propagationTime` were introduced. Together, these values describe a linear increase and decrease over time, a simplified version of the Fire Development Curve 2.2.3. The propagation time is defined by `propagationTime`, the initial temperature by `initialBurningTemperature` and the linear change per second by `burningTemperatureChangePerSecond`. The initial temperature is increased over half the propagation time and decreased over the second half, resulting in a final value equal to `initialBurningTemperature`.

FireComponent and FireState

Similar to the previously described approach the `FireComponent` class is attached to entities and describes their propagation behavior. It only holds constant data, while the `FireState` class has an internal state that changes over the course of the propagation.
As in the first approach, all `FireComponent` instances are stored within a spatial index grid implemented by the `SpatialGrid` class. For this approach, `FireComponent` instances are represented by capsules within the grid. The reason for this is a more accurate representation of objects. Furthermore, the `FireComponent` contains a reference to a `FireBurnSettings` instance as a member. Contrary to the parts of the system described in the following sections, the `FireState` class does not have a temperature that is being simulated by the Fire Propagation System.
Its temperature threshold is reached when the simulated temperature of one of the `FireGridCell` instances it is intersecting with reaches this value. The internal state of a the `FireState` class is defined by two members:

- `isOnFire`
- `isBurntDown`

These two flags are set by `FireHeatSource` class. The `FireState` class can be in the following internal states:

- `Default`
  `isOnFire == false && isBurntDown == false`
- `OnFire`
  `isOnFire == true && isBurntDown == false`
- `BurntDown`
  `isOnFire == false && isBurntDown == true`

FireGridCell

Similar to the object-based approach 4.2, this approach uses a three-dimensional grid as underlying structure for the propagation. Several requirements were defined for this grid:

- All entries within this grid are of the same fixed size.
- Each axis value of an entry's position has to be a multiple of the grids' size.
- Each grid position is unique and can only be occupied by a single entry.

These requirements were put on top of spatial hash implementation of the SpatialGrid class. Entries of this grid are `FireGridCell` instances. They are *Heat Receivers* and share the same instance of the `FireBurnSettings` class globally, resulting in the same propagation behavior for each `FireGridCell` instance. While the `FireGridCell` class is not attached to an entity, it knows about `FireComponent` instances that intersect with its cell bounds. The temperature of the `FireGridCell` class is simulated by the Fire Propagation System (`myTemperature`). In this project, the environment temperature of the game world was used as initial temperature. This temperature increases when a `FireGridCell` class instance is being heated by a `FireHeatSource` class instance. As do Fire States, each `FireGridCell` instance has an internal state represented by three members:

- `isOnFire`
- `isBurntDown`
- `isCoolingDown`

These internal states are defined as followed:

- `Default`
  `isOnFire == false && isBurntDown == false && isCoolingDown == false`
- `OnFire`
  `isOnFire == true && isBurntDown == false && isCoolingDown == false`
- `BurntDown`
  `isOnFire == false && isBurntDown == true && isCoolingDown == false`
- `CoolingDown`
  `isOnFire == false && isCoolingDown == true`

When the simulated temperature of a `FireGridCell` instance reaches the value defined by `temperatureThreshold` of the shared `FireBurnSettings` class instance, it is considered `OnFire`. Starting from this moment, it stays in this state for the amount of seconds defined by `propagationTime`. After this time, the `FireGridCell` class

transitions in the `BurntDown` state. The `CoolingDown` state contradicts the idea of exclusive entity states. If a `FireGridCell` instance is not heated by a FireHeatSource class instance, it transitions into the `CoolingDown` state and its temperature decreases until it reaches the environment temperature. This is independent of the`BurntDown` state, which essentially means the `FireGridCell` class can be in the `CoolingDown` and `BurntDown` state at the same time. This is due to the fact that a `FireGridCell` can still receive thermal energy, even though it is already burnt. Furthermore, the internal state of a `FireGridCell` instance does not necessarily reflect the internal states of intersecting `FireState` instances. Since they reference a different instance of the `FireBurnSettings` class, the value of `temperatureThresholds` might not have been reached by the simulated temperature of the `FireGridCell` instance.
Furthermore, the `FireGridCell` class keeps track of all `FireHeatSource` instances it currently receives thermal energy from. When a `FireGridCell` instances does not have any `FireHeatSource` instances it receives thermal energy from, it transitions to the `CoolingDown` state and its temperature starts decreasing. This happens by using formula described in 4.3.1.
A `FireGridCell` instance is not created when either the dedicated server process is starting up or a player joins. Their creation is triggered when a `FireHeatSource` instance is querying the `SpatialGrid` instance at a position no `FireGridCell` instance exists yet. The lifetime of each `FireGridCell` instance is very limited, mostly for performance and memory reasons. When in the `CoolingDown` state and the temperature of the `FireGridCell` instance is equal to the current environment temperature, the instance will be removed from the spatial hash.

FireHeatSource

The `FireHeatSource` class fulfills the role of a *Heat Source* transferring thermal energy to *Heat Receivers* like the `FireGridCell` class. As for the `FireGridCell` class, the temperature of `FireHeatSource` class is being simulated.
One of the members of the `FireHeatSource` class is a reference to a `FireBurnSettings` instance describing its temperature development.
The `FireHeatSource` class exclusively transfers thermal energy to the `FireGridCell` class, which indirectly notifies the `FireState` class when the cell temperature reaches the value of its `temperatureThreshold` member. The creation of a `FireHeatSource` class instance can be triggered by multiple events:

- player actions,
- scripts,
- a `FireGridCell` instance changes its internal state to `OnFire`,
- a `FireState` instance changes its internal state to `OnFire`

While the first two items depend on actions that happen outside the system, the last two are triggered by the Fire Propagation System itself. The simulated temperature of the `FireHeatSource` class entirely depends on its `FireBurnSettings` instance member which it inherits from its origin. The `FireHeatSource` class has one of three possible origins:

- `None`
- `Cell`
- `Object`

The time a `FireHeatSource` instance actively transfers thermal energy is defined by the `propagationTime` member of the `FireBurnSettings` reference of its origin. Before it is destroyed, it triggers a state transition to `BurntDown` for its origin. The `myPosition` member of the `FireHeatSource` class is inherited by the position of its origin. The `radius` member, which determines the intersecting `FireGridCell` class instances is set depending on the origin defined by the `FireHeatSourceOrigin` enumeration:

- `None`
  The radius and position is defined in the event that triggered the creation of this `FireHeatSource`.
- `Cell`
  The position of the `FireGridCell` instance is inherited, the `radius` is equal to the size of a `FireGridCell`, which guarantees intersection with all its neighbors.
- `Object`
  The position of the `FireState`, to be more precise the entity the `FireState` instance is attached to, is inherited. While `radius` is determined by the `radius` of the capsule of the `FireComponent` instance this instance is referencing.

Rules of Propagation

In this concept, the propagation is based on two `SpatialGrid` instances. While one instance contains all `FireGridCell` instance, the other stores `FireState` instances. Entries of both instances store their position in the grid and in the game world. The exponential growth of a fire is visualized by Figure 4.10. While an initial `FireHeatSource` that is triggered from outside the system can affect a variable number of `FireGridCell` instance depending on its center position and its radius, each `FireHeatSource` instance triggered by the state change of a `FireGridCell` instance will only affect its eight neighbors and the `FireGridCell` instance referenced as origin.
A `FireGridCell` instance can have an unlimited amount of `FireHeatSource` instances transferring thermal energy to it and vice versa. The amount of thermal

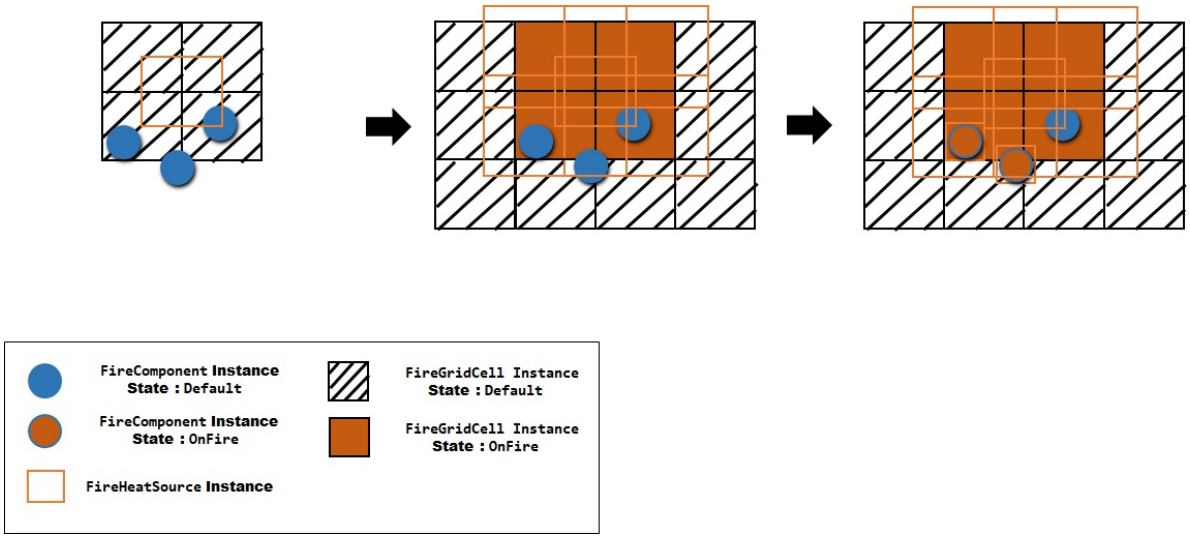

FIGURE 4.10: Fire Propagation Cells

energy transferred to a `FireGridCell` instance by a single `FireHeatSource` instance. Although, the total amount of thermal energy transferred to a `FireGridCell` rises with the amount of `FireHeatSource` instances heating it.

Terrain Position As the terrain in game worlds is rarely entirely flat, a 3-dimensional grid is needed. To prevent situations were thermal energy is transferred to `FireGridCell` instances not intersecting with the terrain, the y-coordinate (Up) for each `FireGridCell` instance is determined by a ray cast in y-direction. The y-coordinate of the hit point is rounded to the closest multiple of the cell size. This combination of x-, y- and z-coordinate defines the position of each `FireGridCell` instance for the spatial hashing. `FireGridCell` instances are only created when they intersect with the terrain. The result of this process is visualized in figure 4.11.
If the sphere defined by the position and twice the radius of the `FireHeatSource` instance spawned by an external event does not intersect with the terrain mesh, it will immediately be destroyed. An example of this distance check can be seen in figure 4.12.

Visual Representation

The visual representation is limited to `FireHeatSource` instances with origin `Cell` and is handled by the system on client-side only. When a `FireHeatSource` instance with origin `Cell` is created on client-side, the system spawns a particle effect at its position. Its life cycle depends on the `FireHeatSource` instance, as it will be destroyed when the instance is destroyed. Since each `FireGridCell` instance can only be set on fire once during its life cycle and thus only spawn a single `FireHeatSource` instance, it is guaranteed that multiple particle effects can not spawn at the same position.
As described in 4.3.1, the terrain position of each `FireGridCell` instance has to be

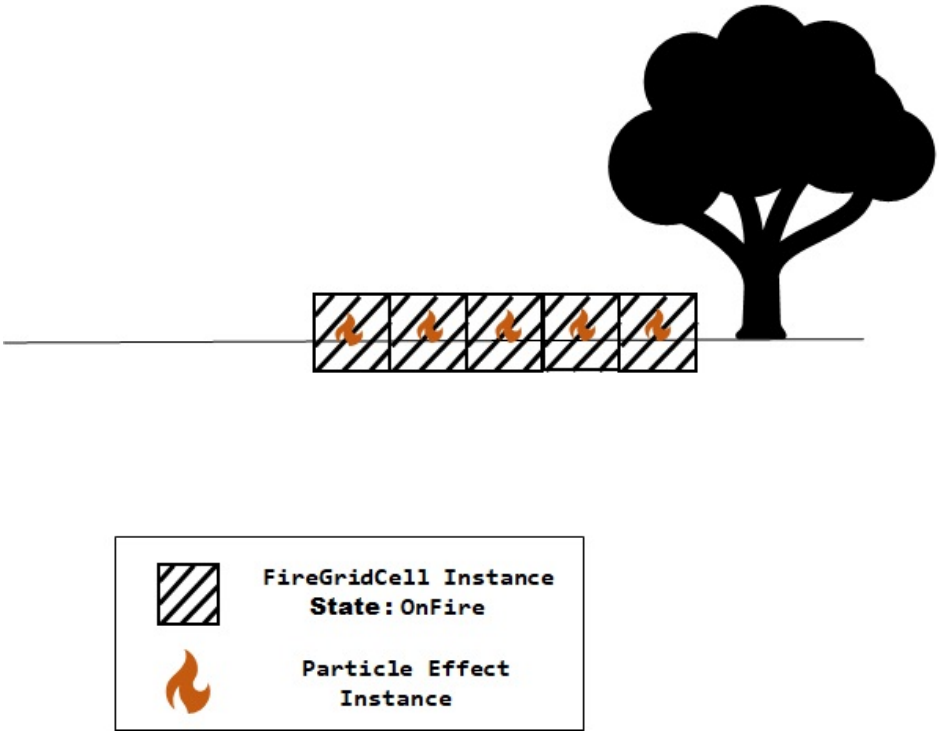

FIGURE 4.11: Fire Propagation On Terrain

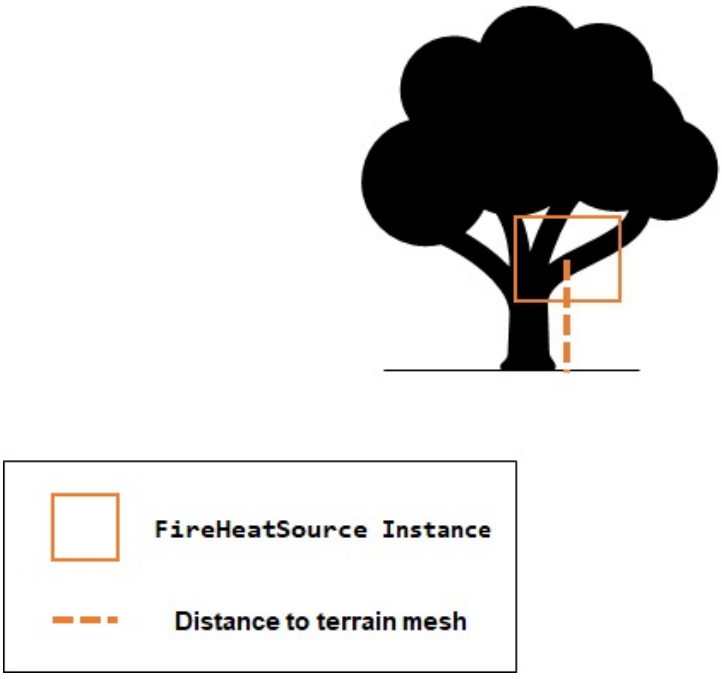

FIGURE 4.12: Fire Propagation On Terrain

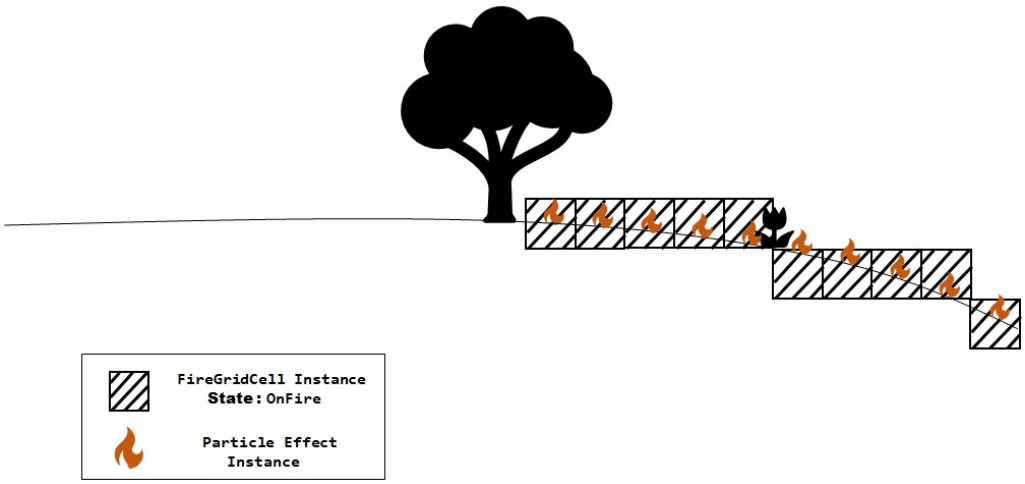

FIGURE 4.13: Fire Propagation Particles

determined by ray cast. While the cell center itself will be rounded to the nearest multiple of the cell size, the actual y-coordinate of the ray hit will be used as position for the particle effect spawned for this instance. Visualized in figure 4.13, the position of a particle effect will always be within a cell, but does not correspond with its center.

Water Manager

To prototype interaction with another system, a `WaterManager` class together with a basic representation of temporary water volumes, the `WaterWetArea` class was introduced. The `WaterWetArea` class is defined by a position, a radius and a lifetime. All active `WaterWetArea` class instances are stored in an array. The idea is to allow communicate the creation of `WaterWetArea` class instances to the Fire Propagation System and remove all active fires in this area.
On the other hand, the Fire Propagation System has to query active `WaterWetArea` instances before a fire can propagate further.

4.3.2 Implementation

This implementation was done in C++ within *Snowdrop*. It will be presented in the same way as the object-based approach.

Structure

With `Shared_`, `Client_` and `Server_`, the same prefixes are used for this approach. If no prefix is given, it is always the base class that is being referred to. The structure of this approach is similar to the object-based approach, it was developed for the same project.An overview of the structure of this implementation is given by figure 4.14. Both, the `FireComponentManager` and `FireManager` class have almost the identical purpose as for the previous approach. The general purpose and functionality of the

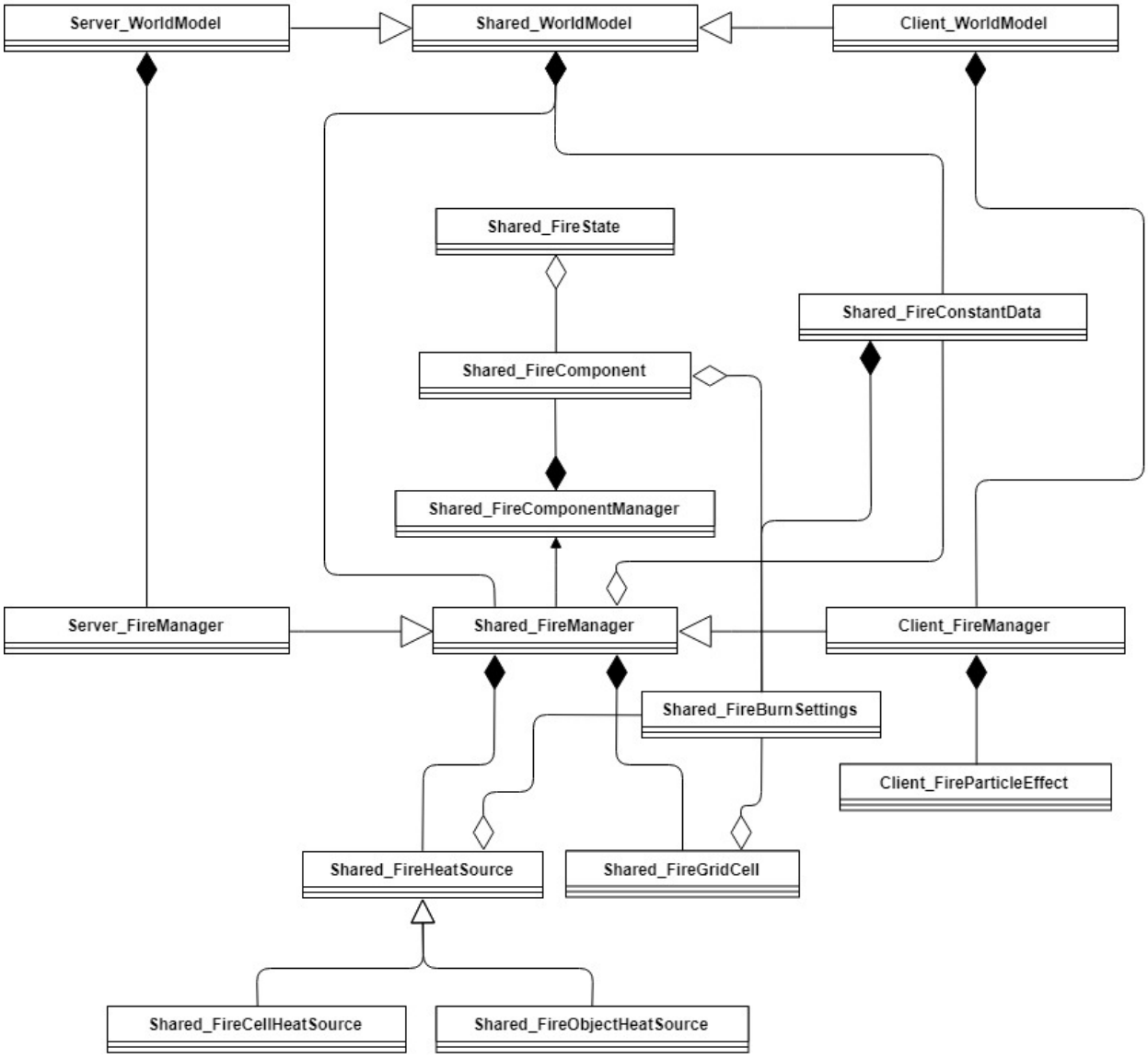

FIGURE 4.14: Implementation Overview

`FireComponent` and `FireState` class have not changed, only member variables have been modified to fit the concept. This will be discussed in more detail in the following sections. The newly introduced classes `FireGridCell` and `FireHeatSource` are owned and managed by the `FireManager` class. Both classes do not have server- or client side specific implementations. Although the `FireHeatSource` class, with the addition of the `FireCellHeatSource` and the `FireObjectHeatSource`, does have two derived classes. For all constant data that is used within the Fire Propagation System, the `FireConstantData` class was introduced. It provides centralized access to constant data.

Network Overview

Since the network communication and the general functionality of network events is part of *Snowdrop* and not the developed system, it does not differ from the implementation of the object-based approach. An overview of the network-related parts of the system is given by figure 4.15. While the `FireManagerSpawnNetworkEvent` class is reused for this approach, the `FireManagerUpdateNetwork` class is removed. This is mainly due to project development circumstances, as the replication of entity state transitions is now handled by the `EntityStateComponent` class itself. The `FireManagerSpawnNetworkEvent` class is used to sync the creation of new `FireHeatSource` instances. Depending on the origin, different members are used. For a `Cell` origin, the `myPosition` member of the event determines which `FireGridCell` is the origin. This is possible due to the fact that each `FireGridCell` instance has an unique position. For `Object`, a UID of a `FireComponent` instance is sent over to determine which `FireState` instance should be referenced. For `None`, only a UID that can be resolved to a `FireBurnSettings` instance by the `FireConstantData` class is needed.

FireBurnSettings

The `FireBurnSettings` class describes the propagation behavior of a flammable object. It is used to determine when an internal state transition of a `FireState` or a `FireGridCell` instance is supposed to happen and how a `FireHeatSource` instance triggered by this state transition should behave. In addition to the members described in 4.3.1, the implementation (figure 4.16) features a name `myName` which was mostly used for debugging purposes and a UID `myUID`, which uniquely identifies each `FireBurnSettings` instance. An overview of this implementation is provided by figure 4.16

FireConstantData

The `FireConstantData` class was introduced to minimize memory consumption. It reads from data files when a client- or server process is started and is deleted when the process is shut down. It is important to note that an instance of this constant

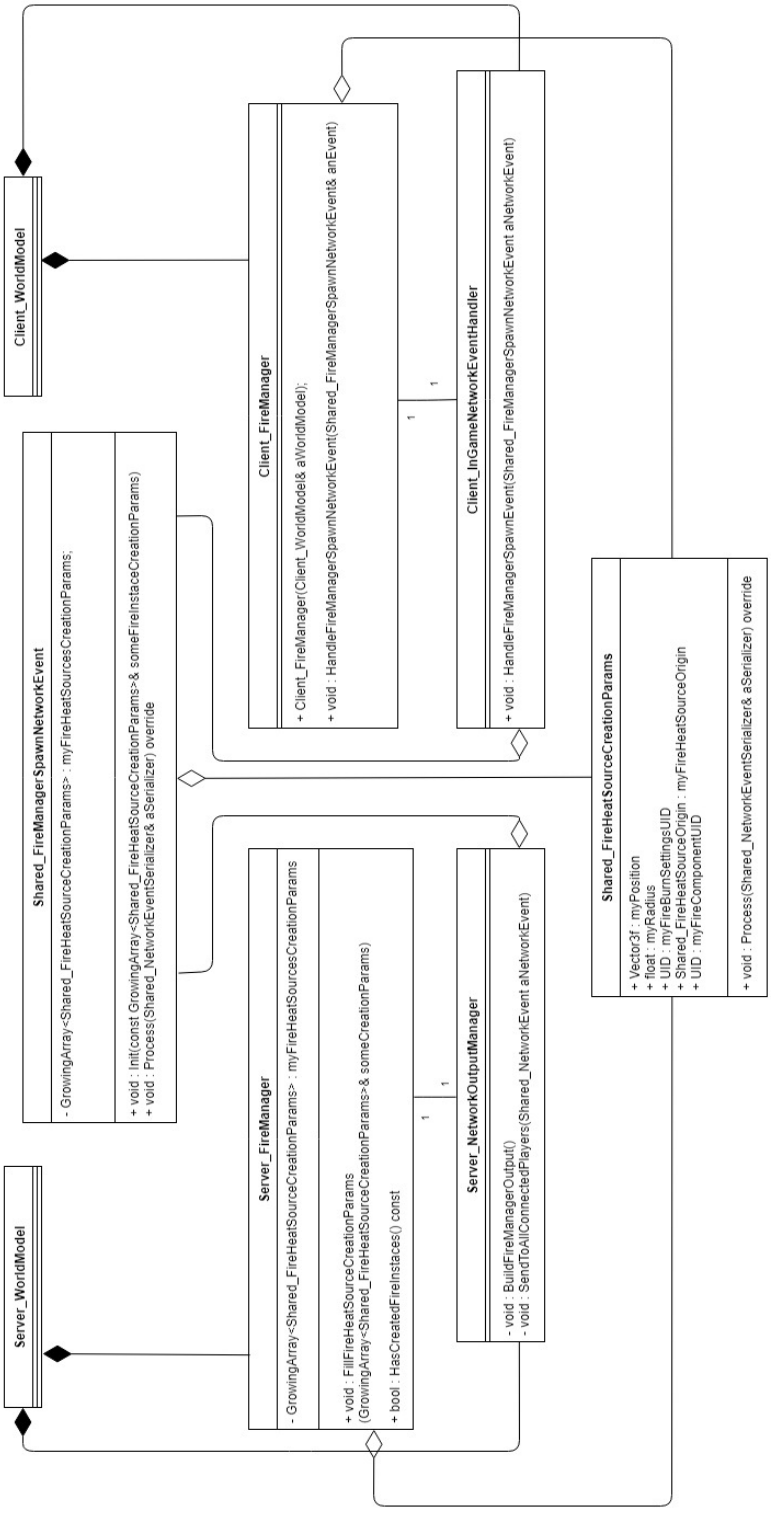

FIGURE 4.15: Network Overview

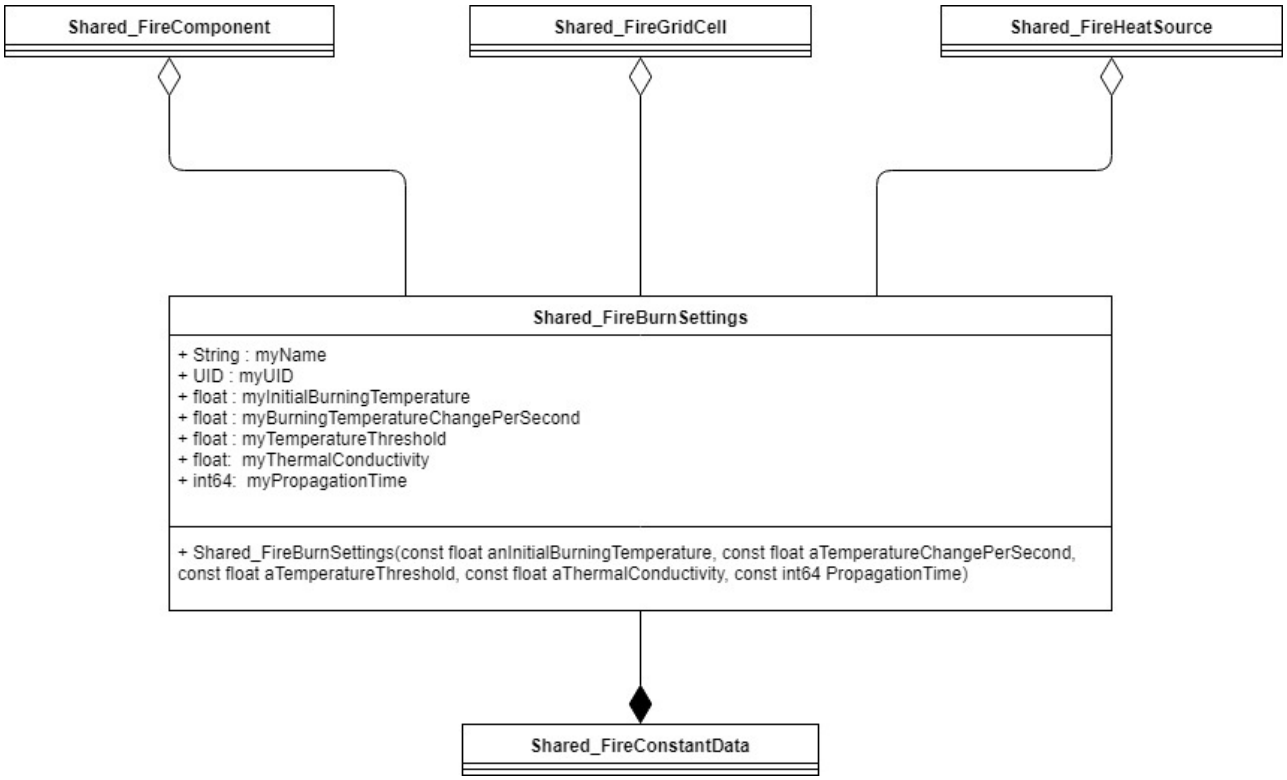

FIGURE 4.16: `FireBurnSettings` UML

data exists for the server and each client. An overview of the implementation is given by figure 4.17.

To save memory, this class holds a list of all existing `FireBurnSettings` instances in its member `myFireBurnSettingsList`. The `myGlobalFireBurnSettings` member is a pointer to a `FireBurnSettings` instance that is not part of `myFireBurnSettingsList`. It is used for all `FireGridCell` instances in the game world.

`myFireParticleEffectObjectTemplate` stores a template for the particle effect that will be instantiated at the position of `FireGridCell` instances in the `OnFire` state.

`myStatusEffectUID` and `myFireDamageStatusEffect` both refer to a `StatusEffect` instance that is responsible for dealing damage to the player and NPCs. All three members will not be further discussed in this section, as they are all related to other systems within *Snowdrop*.

The value of `myGlobalCellSize` is used to define `FireGridCell.mySize` for all instances. `myDefaultWorldTemperature` is used as the initial value of `FireGridCell.myTemperature` if not overridden by the current environment temperature. Finally, `myMinTemperatureChangePerSecond` defines the minimum change in temperature of a `FireGridCell` instance in the `CoolingDown` state.

`myGlobalFireBurnSettings` allows `FireHeatSource` and `FireGridCell` instances to simply reference a `FireBurnSettings` instance instead of owning it. Based on the same principle, `myStatusEffectUID` is resolved by the `StatusEffectManager` class to a pointer to a `StatusEffect` instance. The `StatusEffect` modifies attributes of NPCs or the player. In case of the Fire Propagation System, it deals

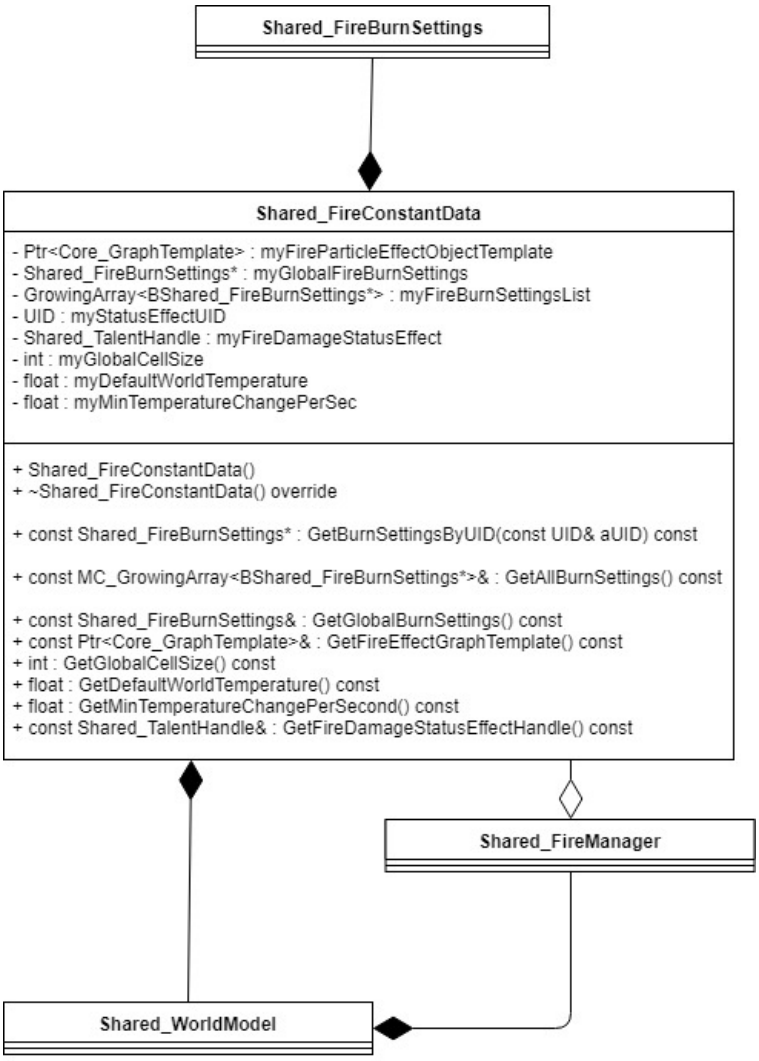

FIGURE 4.17: `FireConstantData` UML

damage over time. It is important to note that, while a `FireBurnSettings` instance can be created during runtime, `myFireBurnSettingsList` is a private member to which the `FireConstantData` class does not provide write access. Each `FireBurnSetting` instance has to be defined in data files. The `FireBurnSetting` instance `myGlobalFireBurnSettings` points to is used for each created `FireHeatSource` with origin `Cell`. Furthermore, this is the only instance of the `FireBurnSettings` class that is guaranteed to exist.
Within the `GetBurnSettingsByUID()` function, the `myGlobalFireBurnSettings` member is returned when the UID passed in as argument does not match with the `myUID] member of a \inline{FireBurnSettings` instance within `myFireBurnSettingsList`. For all other members accessors are provided as displayed in 4.17.

FireComponent

The members of the `FireComponent` class were, compared to the previous approach, modified to fit this concept. As the `FireBurnSettings` class was introduced as structure to combine all necessary propagation properties, only a reference to an instance in form of `myBurnSettings` is needed. While the spatial hash implementation of the `SpatialGrid` class is still used, all `FireComponent` instances are now represented by a capsule within the grid (`myFireGridCapsule`). The members `myLink`, `myEntityId` and `myPosition` have been reused from the first approach. As there is no dedicated member for the radius anymore, it has to be computed based on the extents of `myFireGridCapsule`. Althought being represented by capsules, the `SpatialGrid` class uses spheres to determine the spatial index or indices a `FireComponent` is mapped to. `myPosition` describes the center of the capsule, while the radius is the minimum to describe a sphere that completely encapsulates the capsule.

FireState In this concept, the internal state of the `FireState` class is represented by two boolean members which were explained in section 4.3.1. `FireComponent.myBurnSettings` referenced in `myFireComponent` provides all necessary data when a `FireState` instance's internal state transitions to `OnFire` and triggers the creation of a `FireHeatSource` instance.

FireComponentManager

The implementation of the `FireComponentManager` class for this approach provides a similar functionality as the previous implementation in the object-based approach, it is visualized by figure 4.19
As the member variables were not modified, they will not be listed again in this section.
To provide access to the `SpatialGrid` instance all `FireComponent` instances are stored

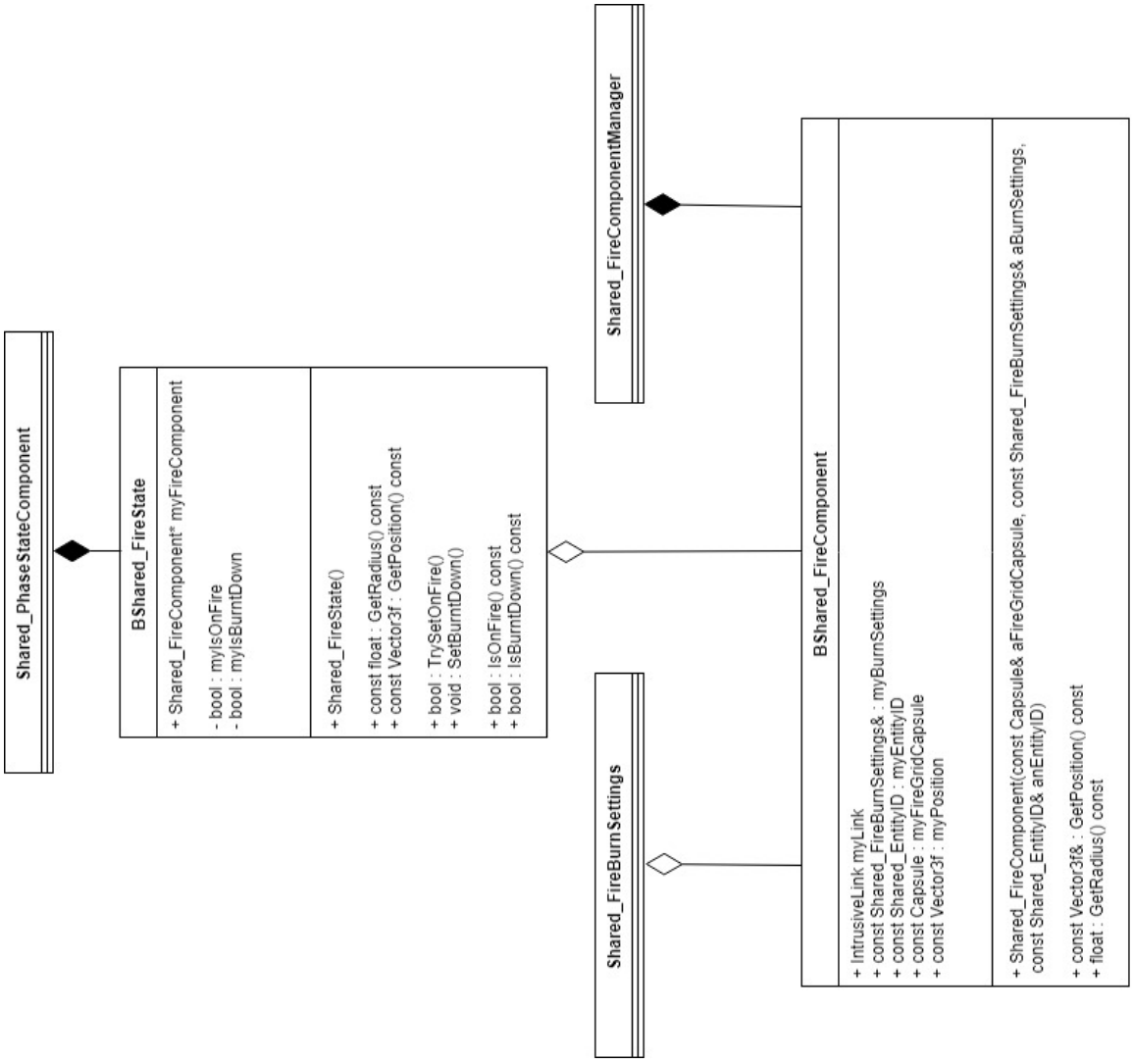

FIGURE 4.18: `FireComponent` UML

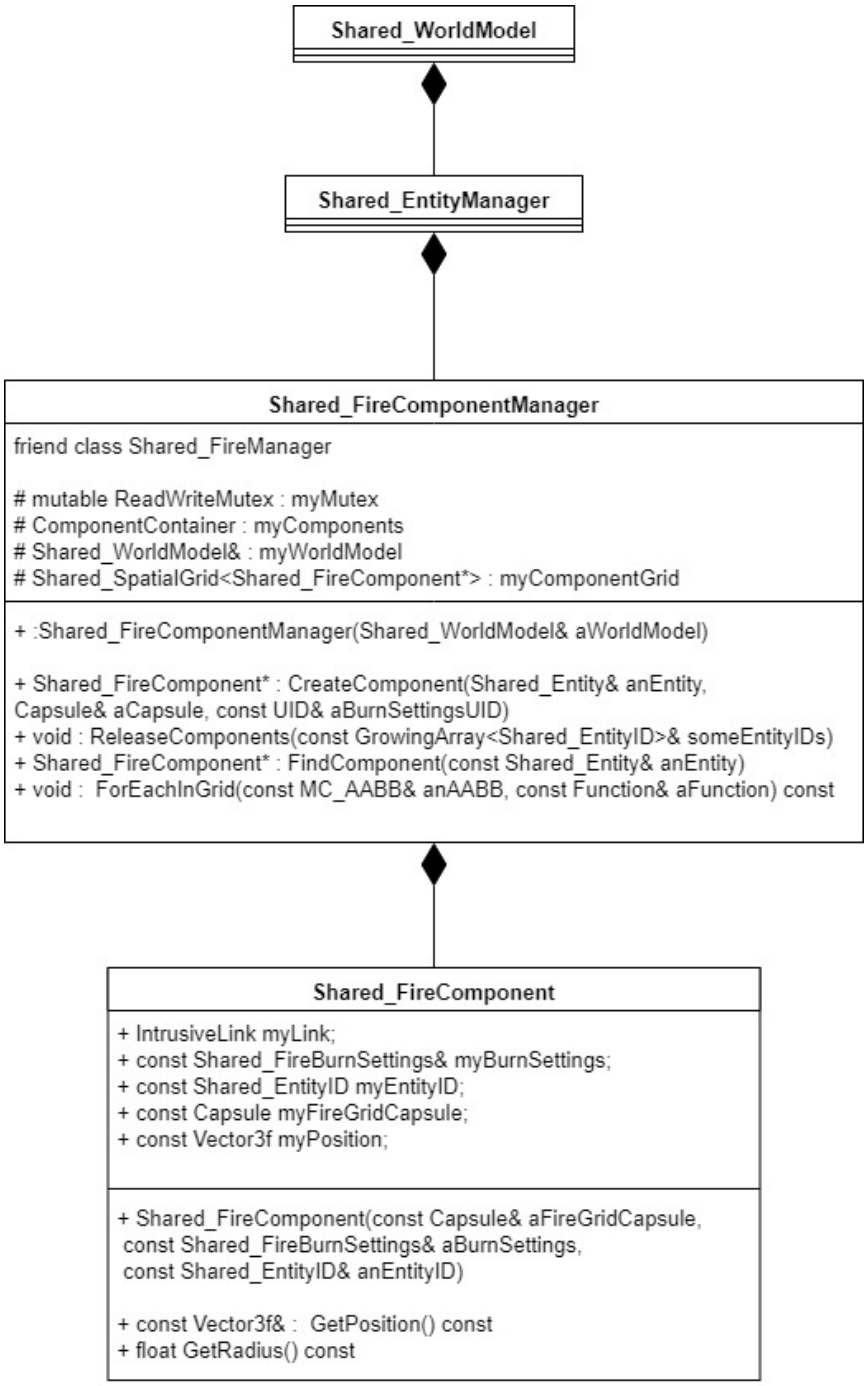

FIGURE 4.19: `FireComponentManager` UML

in, the `ForEachInGrid()`(6) function was added. This function allows to run specific functions on `FireComponent` instances in a defined area.

Function 6: `FireComponentManager::ForEachInGrid()`

```
Input: anAABB, aFunction
Output: No output
for (each fireState in fireStateGrid)
{
    if (position of fireState is within AABB)
    {
        execute aFunction with fireState as argument
    }
}
```

Fire Manager

While the role of the `FireManager` class in the system has not changed much, the implementation differs in many aspects from the previous approach. An overview of these can be seen in figure 4.20.
While the communication between the `FireManager` class, the `WorldModel` class and the `FireComponentManager` class remains the same for the most part, communication with new systems has been introduced.
The `EnvironmentManager` class handles the weather simulation of the game world. The `myEnvironmentManager` member stores a reference to an instance of this class. It holds data about the current weather in the game world. For this implementation, it was used to retrieve the current temperature in the game world. As its implementation is not part of the system, it will not be discussed in this thesis.
A reference to a `FireConstantData` class is introduced as the member `myFireConstantData` due to the fact that the `FireManager` class uses most of the data for the simulation. The implementation of this concept required two separate `SpatialGrid` instances. While the instance that stores the `FireComponent` instances has only changed very little, a new spatial hash with `FireGridCell` instances was added. It is stored in the `myFireCellGrid` member variable. Although the needed functionality could have been provided by a hash map, with a key value pair of a unique position and a `FireGridCell` instance, a spatial index grid was chosen as all its functionality already existed within *Snowdrop*. During development many functions that are be called from multiple threads were identified. While access from other systems is limited to creating `FireHeatSource` instances, removing `FireComponent` and `FireState` instances from the propagation and trigger a state transition for `FireHeatSource` instances to `BurntDown` manually, the life cycle of a `FireHeatSource` demands accessing and modifying the three main data structures within the class:

- `myUpdatedHeatSources`,
- `myCoolingDownGridCells`,

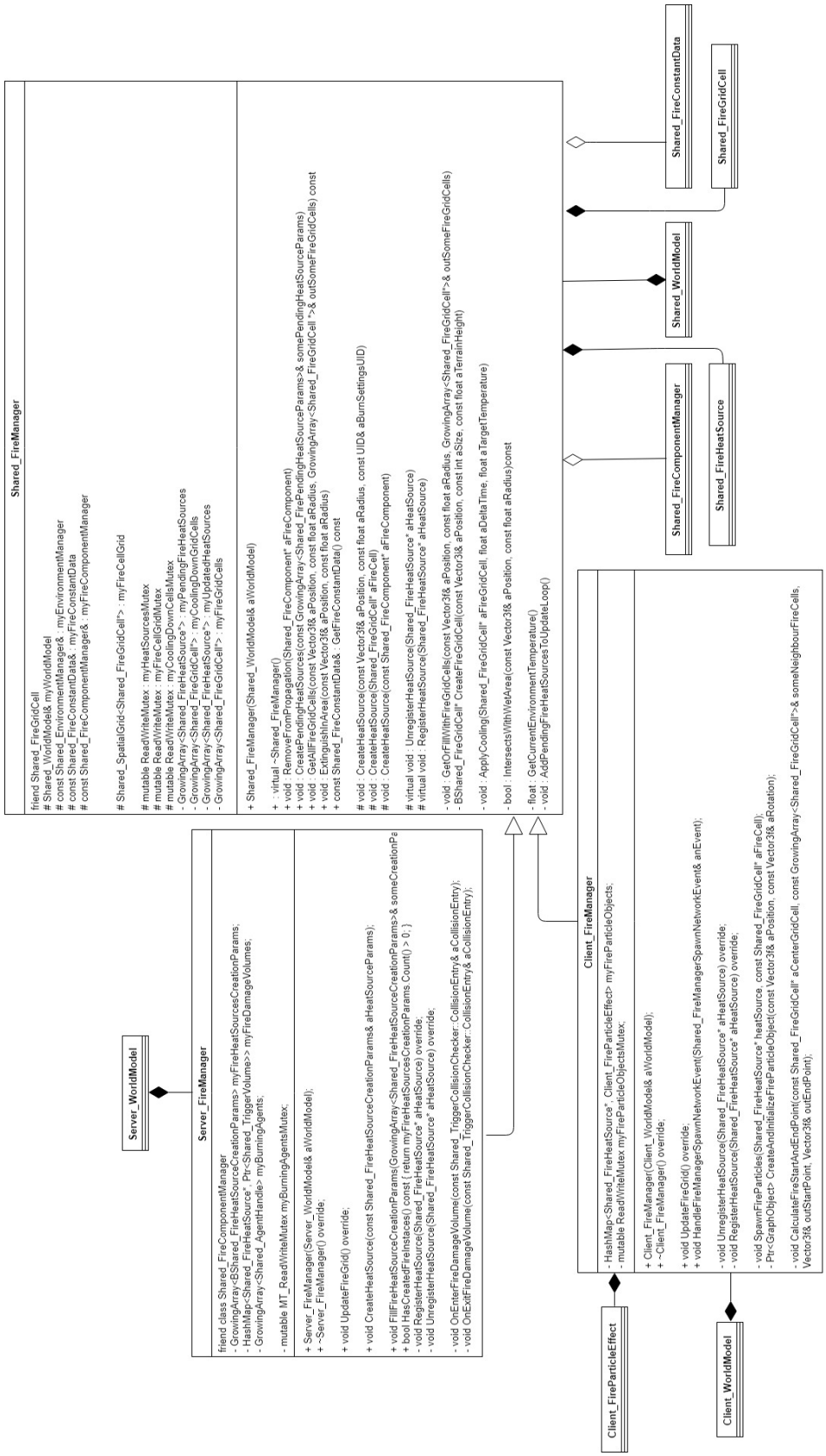

FIGURE 4.20: `FireManager` UML

- `myFireCellGrid`

Although there is no technical reason to use more than one mutex, it communicates the reason why a mutex is locked for a specific part of a function more clearly, as the name of the member variable was chosen accordingly. It also decreased the scope in which mutex-locking is required, resulting in less potentially contention.
The `FireManager` class provides three overloaded versions of the `CreateHeatSource()` function, one per value in the `FireHeatSourceOrigin` enumeration (`None`, `Cell`, `Object`). In this implementation, two derived classes of `FireHeatSource` are introduced. `FireCellHeatSource` and `FireObjectHeatSource`. These will be discussed in more detail in a later section. Each derived class is associated with a value of `FireHeatSourceOrigin`:

- None
  `FireHeatSource`, typically triggered from external events.
- Cell
  `FireCellHeatSource`, is created when a `FireGridCell` instance catches fire.
- Object
  `FireObjectHeatSource`, is created when a `FireState` instance catches fire.

The general structure of all overloaded versions of the `CreateHeatSource()` function is similar, while their inputs differ. A `FireHeatSource` instance's area of effect is defined by a sphere which is retrieved from different sources for each function. The function `CreateHeatSource()` creates a `FireHeatSource` instance with `myOrigin == FireHeatSourceOrigin.None`. Position, radius and a UID for a `FireBurnSettings` instance are explicitly passed in as arguments.
The sphere defined by position and radius is used to determine intersections with `WaterWetArea` and `FireGridCell` instances. Intersections with any number of either of them lead to an immediate return of the function, and thus no creation of a `FireHeatSource` instance. Through the `FireConstantData` class, a pointer to an existing `FireBurnSettings` instance can be retrieved by using the passed in UID.

According to the defined requirements (4.3.1) on top of the spatial hash implementation storing the `FireGridCell` instances, the `GetOrFillWithFireGridCells()` (8) function was constructed. Firstly, a AABB is constructed out of the parameters position and radius. The resulting minimum and maximum position is then passed on to the `SnapToFireGridCell()` function, which snaps the value of each axis to the closest multiple of `FireConstantData.myGlobalCellSize`. A nested for-loop then iterates over all the x- and z- axis values between the minimum and the maximum position incremented by `FireConstantData.myGlobalCellSize` each

Function 7: `FireManager::CreateHeatSource()`

```
Input: aPosition, aRadius, aBurnSettingsUID
Output: No output
if (sphere defined by aPosition and aRadius intersects with an active WaterWetArea
 instance)
{
   return
}
WriteLock(fireCellGridMutex)
GetOrFillFireGridCells(aPosition, aRadius, gridCellArray)
Unlock(fireCellGridMutex)
if (gridCellArray is empty)
{
   return
}
burnSettings = myFireConstantData.GetBurnSettingsByUID(aBurnSettingsUID)
if (burnSettings equals NULL)
{
   burnSettings = myFireConstantData.GlobalBurnSettings()
}
fireHeatSource = new FireHeatSource(aPosition, aRadius, aBurnSettingsUID,
 gridCellArray, origin = None)
RegisterHeatSource(fireHeatSource)
```

loop. This guarantees that each possible `FireGridCell` instance within the specified area is only visited once. A ray casted in y-direction determines the intersection with the terrain mesh. The y-axis value is then snapped to the closest multiple of `FireConstantData.myGlobalCellSize`. To prevent `FireGridCell` instances being created too far away from the actual position passed to the function, a radius check is performed. If the distance between the y-coordinate of the ray hit and the y-coordinate of the position handed to the function is bigger than radius, no `FireGridCell` instance is created. If the difference is smaller, already existing `FireGridCell` instances that were queried from `myFireGrid` are checked for each position consisting of x- and z-coordinate of the loop and the determined y-coordinate of the terrain mesh. If a `FireGridCell` already exists at this position, it will be added to the returned array, if not, a new `FireGridCell` will be created at this position. This ensures that each position of a `FireGridCell` instance is unique.

For the creation of a new `FireGridCell` instance, the following parameters are needed:

- position,
- size,
- list of intersecting `FireComponent` instances

While the position and the size is passed to the `CreateFireGridCell()` (9) function, the intersecting `FireComponent` instances are determined within the function by querying `FireComponentManager.myFireStateGrid`. The `UpdatedFireGrid()` function loops through `myupdatedHeatSources` and `myCoolingDownGridCells`.

Function 8: `FireManager::GetOrFillWithFireGridCells()`

```
Input: aPosition, aRadius, outFireGridCells
Output: No output
create extents 3-dimensional vector out of aRadius
minPos = extents + aPosition
minPos = extents - aPosition
SnapToFireGridCell(minPos, maxPos)
AABB = new AABB(minPos, maxPos)
get all fireGridCells that intersect or are within AABB from fireCellGrid
for (each multiple of cellSize between minPos and maxPos in the x-axis)
{
    for (each multiple of cellSize between minPos and maxPos in the z-axis)
    {
        get y-position by casting ray in y-axis direction and taking
         y-coordinate of hit point
        SnapToFireGridCell(y)
        if (y − position.y < radius)
        {
            for (each fireGridCell in fireGridCells)
            {
                if (fireGridCell.myPosition is equal to Vector3(x,y,z))
                {
                    outFireGridCells.Add(fireGridCell
                }
                else
                {
                    CreateFireGridCells(aPosition,
                     FireConstantData.myGlobalCellSizecellSize,y)
                    add newly created FireGridCell to outFireGridCells
                }
            }
        }
    }
}
```

Function 9: `FireManager::CreateFireGridCell()`

```
Input: aPosition,aSize, aTerrainHeight
Output: A pointer to fireGridCell
extents = new Vector3(aSize * 0.5)
minPos = extents + aPosition
minPos = extents - aPosition
AABB = new AABB(minPos, maxPos)
Query fireComponentGrid to find all fireComponents intersecting with
 AABB
fireGridCell = new FireGridCell(aPosition, aSize, fireComponents)
fireCellGrid.Insert(fireGridCell)
return fireGridCell
```

It updates the propagation state by calling `FireHeatSource.PropagateHeat()` on each existing class instance and removes them from the propagation once `FireHeatSource.IsPropagating()` returns `false`. This function demonstrates the use of multiple mutexes. Instead of locking a single mutex over the whole function, it is possible to lock only parts of this function, namely the for-loop over `myUpdatedHeatSources` and the for-loop over `myCoolingDownGridCells`, thus reducing the contention time.
Once `FireGridCell.myHeatSources` is empty, the `FireGridCell` instance will be added to `myCoolingDownGridCells`. The `UpdateFireGrid()` (10) function manages the process of `FireGridCell` instances dropping their temperature until they reach the environment temperature by calling the `ApplyCooling()` function on each entry in this list.
At the end of the function, all entries of `myPendingHeatSources` are added to `myUpdatedHeatSources`, this was done to prevent more mutex contention and to limit the write access to `myUpdatedHeatSources` to a single function. As men-

Function 10: `FireManager::UpdateFireGrid()`

```
Input: No input
Output: No output
WriteLock(heatSourcesMutex)
for (each heatSource in updatedHeatSources)
{
    heatSource.PropagateHeat()
    if (heatSource.myIsPropagating == false)
    {
        heatSource.SetInactive()
        UnregisterHeatSource(heatSource)
    }
}
Unlock(heatSourcesMutex)
get deltaTime
get environmentTemperature from GetCurrentEnvironmentTemperature()
WriteLock(coolingDownCellsMutex)
for (each fireGridCell in coolingDownCells)
{
    ApplyCooling(fireGridCell, deltaTime, environmentTemperature)
    if (fireGridCell.myTemperature < environmentTemperature)
    {
        WriteLock(fireCellGridMutex)
        fireCellGrid.Remove(fireGridCell)
        coolingDownCells.Remove8fireGridCell)
    }
}
WriteLock(heatSourcesMutex)
AddPendingHeatSourcesToUpdateLoop()
```

tioned in 3.4, the loading and unloading of Sectors results in `FireComponent` instances being added to and removed from the `FireComponentManager` class. When a `FireComponent` instance is unloaded, it needs to be properly removed from the system as the `FireObjectHeatSource` and `FireGridCell` class store raw

pointers to a `FireComponent` instance. The `RemoveFromPropagation()` function removes all pointers to the `FireComponent` passed in. If a `FireObjectHeatSource` instance with `myFireState.myFireComponent` pointing to the unloaded instance exists, it will be deleted. The `FireGridCell` class keeps a list of pointers to `FireHeatSource` instances that transfer thermal energy to them. With the creation of a new `FireHeatSource` instance, affected `FireGridCell` instances need to be updated. The `RegisterHeatSource()`(11) function calls the `FireGridCell.RegisterHeatSource()` function on each entry in `FireHeatSource.myHeatedFireGridCells` with a pointer to the newly created `FireHeatSource` instance as argument. If it exists in `myCoolingDownGridCells`, the `FireGridCell` instance will be removed from the `myCoolingDownGridCells` member at the end of the function. On the other hand, when a `FireHeatSource`

Function 11: `FireManager::RegisterHeatSource()`

Input: *aFireHeatSource*
Output: `No output`
for (*each fireGridCell in fireHeatSource.myHeatedGridCells*)
{
    fireGridCell`.RegisterHeatSource(`*fireHeatSource*`)`
    `WriteLock(`*coolingDownCellsMutex*`)`
    coolingDownGridCells`.RemoveIfExists(`*fireGridCell*`)`
}

instance is about to be destroyed, the `UnregisterHeatSource()` function is called. It calls the `FireGridCell.UnregisterHeatSource()` function on each element in `FireHeatSource.myHeatedFireGridCells`. If `FireGridCell.myHeatSources` is empty after this function call, the instance will be added to `myCoolingDownGridCells`. Finally, the `FireHeatSource` instance is removed from `myUpdatedHeatSources`. The `ApplyCooling()` (4) function is called for each element in `myCoolingDownGridCells`. It uses the defined formula in 4.1 to calculate the thermal energy transferred to each element. The resulting value is multiplied by the last frame time and has to be smaller or bigger than the `FireConstantData.myMinTemperatureChangePerSecond`, depending on the sign.

Server_FireManager The derived server class `Server_FireManager` extends the functionality of the base class by adding `TriggerVolume` instances to the game world for all `FireCellHeatSource` instances. These will be stored in `myFireDamageVolumes`. The `TriggerVolume` class triggers events when a player or a NPC, enters or exits them. Additionally, the class stores a list of handles to NPCs and players within at least one `TriggerVolume` instance. The derived server class overwrites three functions of its base class:

- `UpdateFireGrid()`
- `RegisterHeatSource()`

Function 12: `FireManager::ApplyCooling()`

Input: *AFireGridCell*, *aDeltaTime*, *aTargetTemperature*
Output: `No output`

```
heatAmount = myFireConstantData.myGlobalThermalConductivity *
 (targetTemperature - currentTemperatur of fireGridCell)
get minTemperatureChangePerSecond from Fire Constant Data
if (heatAmount > 0)
{
    heatamount =
     heatAmount * deltaTime*MAX(heatAmount,minTemperatureChangePerSecond)
}
else
{
    heatamount =
     heatAmount * deltaTime*MIN(heatAmount,minTemperatureChangePerSecond)
}
fireGridCell.UpdateTemperature(heatAmount)
```

- `UnregisterHeatSource()`

In the overridden `UpdateFireGrid()` (13) function, a for-loop iterates over all elements in `myBurningAgents`. It removes invalid handles when the resource it points to is unloaded and applies damage to NPCs pointed to by valid handles. As all derived classes from `FireManager` should provide the same base functionality, the `UpdateFireGrid()` function of the base class is called at the end. In the overridden `RegisterHeatSource()` function, a `TriggerVolume` instance is created per `FireCellHeatSource`. A new key-value pair consisting of a pointer to the newly created `FireCellHeatSource` instance and to the corresponding `TriggerVolume` instance is added to `myFireDamageVolumes`. Additionally, two function pointers are passed to the `TriggerVolume` instance to be called when an NPC enters or exits it. Furthermore, a `FireHeatSourceCreationParams` instance has to be created for the `FireHeatSource` instance the argument points to and then added to the `myFireHeatSourceCreationParams`. This list is eventually copied to `FireManagerSpawnNetworkEvent.myFireHeatSourcesCreationParams` by the `Server_NetworkOutputHandler` class and sent to the client. Finally, the base class function is called.
`UnregisterHeatSource()`, the third and last overridden function, consists of simply two lines of code. It is removing the key-value pair by its key, the pointer to a `FireHeatSource` instance. This key is passed to the function as an argument. At the end of the function, the base class function is called. The called functions when an NPC enters or exits a `TriggerVolume` instance, `OnEnterFireDamageVolume()` and `OnExitFireDamageVolume()`, simply add or remove a handle of the entering NPC to `myBurningAgents`. No checks are performed if a handle to an NPC already exists, as NPCs being in more than one `TriggerVolume` instance should get multiple amounts of damage. Instead of storing the absolute number of `TriggerVolumes` instances each NPC is intersecting with, an additional handle to

Function 13: `FireManager::UpdateFireGrid()`

Input: *aHeatSource*
Output: `No output`
`create` *triggerVolume* `at` *heatSource* `position with radius`
`add` *triggerVolume* `to` *fireDamageVolumes* `with` *heatSource* `as key`
`add function pointers to EnterFireDamageVolume and ExitFireDamageVolume to` *triggerVolume*
`get FireHeatSourceCreationParams from` *heatSource* `and add it to` *fireHeatSourceCreationParams*
`parent.RegisterHeatSource()`

this NPC is added to the `myBurningAgents`. Before each access to `myBurningAgents`, the `myBurningAgentsMutex` is locked.

Client_FireManager The derived client class of the `FireManager` class handles the visual representation of the Fire Propagation System. It manages the life cycle of particle effects spawned for each existing `FireCellHeatSource` instance and replicates `FireHeatSources` instances out of network events it receives. All active particle effects are stored in `myFireParticleObjects`, while the mutex `FireParticleObjectsMutex` is locked before accessing this member. The class `Client_FireParticleEffect` contains an instance of a particle effect, the implementation of a particle effect within *Snowdrop* will not be discussed in this thesis.
The `HandleFireManagerSpawnNetworkEvent()` function creates new `FireHeatSource` instances out of the received `FireManagerSpawnNetworkEvent` instances. It calls the `CreateHeatSource()` function of the base class for each instance. Which overloaded function is called depends on the value of `myFireHeatSourceOrigin` of each entry in `FireManagerSpawnNetworkEvent.myFireHeatSourcesCreationParams`. If necessary, either a pointer to a `FireGridCell` instance has to be retrieved from `myFireCellGrid` based on `FireHeatSourceCreationParams.myPosition`, or a pointer to a `FireComponent` instance retrieved from the `FireComponentManager` class by calling its member function `FindComponent()`.
Similar to `Server_FireManager.myFireDamageVolumes`, `myFireParticleObjects` is a hashmap with a pointer to a `FireHeatSource` instance as key, and a `Client_FireParticleEffect` instance as value. The overridden `RegisterHeatSource()` function creates a `Client_FireParticleEffect` instance and adds a key-value pair with a pointer to a `FireHeatSource` instance to `myFireParticleObjects`. In the developed prototype, only `FireCellHeatSource` instances are visually represented in the game world. The base function is called in the end.
The overridden `UnregisterHeatSource()` function simply removes the key-value pair from `myFireParticleObjects` according to the `FireHeatSource` instance passed in.

FireGridCell

The `FireGridCell` class represents the only *Heat Receiver* within the system whose temperature is being simulated. It stores `FireHeatSource` instances that it receives thermal energy from in `myHeatSources`, and `FireComponent` instances whose capsule intersect with it in `myFireComponents`. The four internal states discussed in 4.3.1 are represented by three booleans:
`myIsOnFire`,`myIsBurntDown` and `myIsCoolingDown`.
`myMutex` is locking access to `myFireComponents`, which could be accessed by different threads. `myGridPosition` represents a three-dimensional vector specifying the position within `FireManager.myFireCellGrid` and thus each coordinate is a multiple of `mySize`. `myWorldPosition` on the other hand is the actual terrain mesh position used for spawning particle effects.
Most functions of this class are accessors. The `UpdateTemperatureAndGetHeatSourceParams()` (14) function is used by the `FireHeatSource` class to transfer thermal energy to a `FireGridCell` instance and retrieve `FirePendingHeatSourceParams` instances for each `FireGridCell` or `FireComponent` instance where `myTemperatureThreshold` of their referenced `FireBurnSettings` instance is smaller than `myTemperature` of this `FireGridCell` instance.
The `UpdateTemperature()` function multiplies the passed in heat amount with `myBurnSettings.myThermalConductivity` and adds it to `myTemperature`.

Function 14: `FireGridCell:UpdateTemperatureAndGetHeatSourceParams()`

```
Input: aHeatAmount, outPendingHeatSourceParams
Output: No output
UpdateTemperature(heatAmount)
if (myTemperature > myBurnsettings.myTemperatureThreshold and myIsBurntDown
  and myIsOnFire are both false)
{
    add pendingHeatSource with this as origin Cell to
      outPendingHeatSourceParams
}
ReadLock(mutex)
for (each fireComponent in fireComponents)
{
    if (myTemperature > fireComponent.myBurnSettings.myTemperatureThreshold)
    {
        add pendingHeatSource with fireComponent as origin Object to
          outPendingHeatSourceParams
    }
}
```

When a `FireHeatSource` instance or one of its derived classes adds a `FireGridCell` instance to `FireHeatSource.myHeatedFireGridCells`, this `FireHeatSource` instance has to be added to `FireGridCell.myHeatSources` in return. The `RegisterHeatSource()` and `UnregisterHeatSource()` function provide write access to this member.

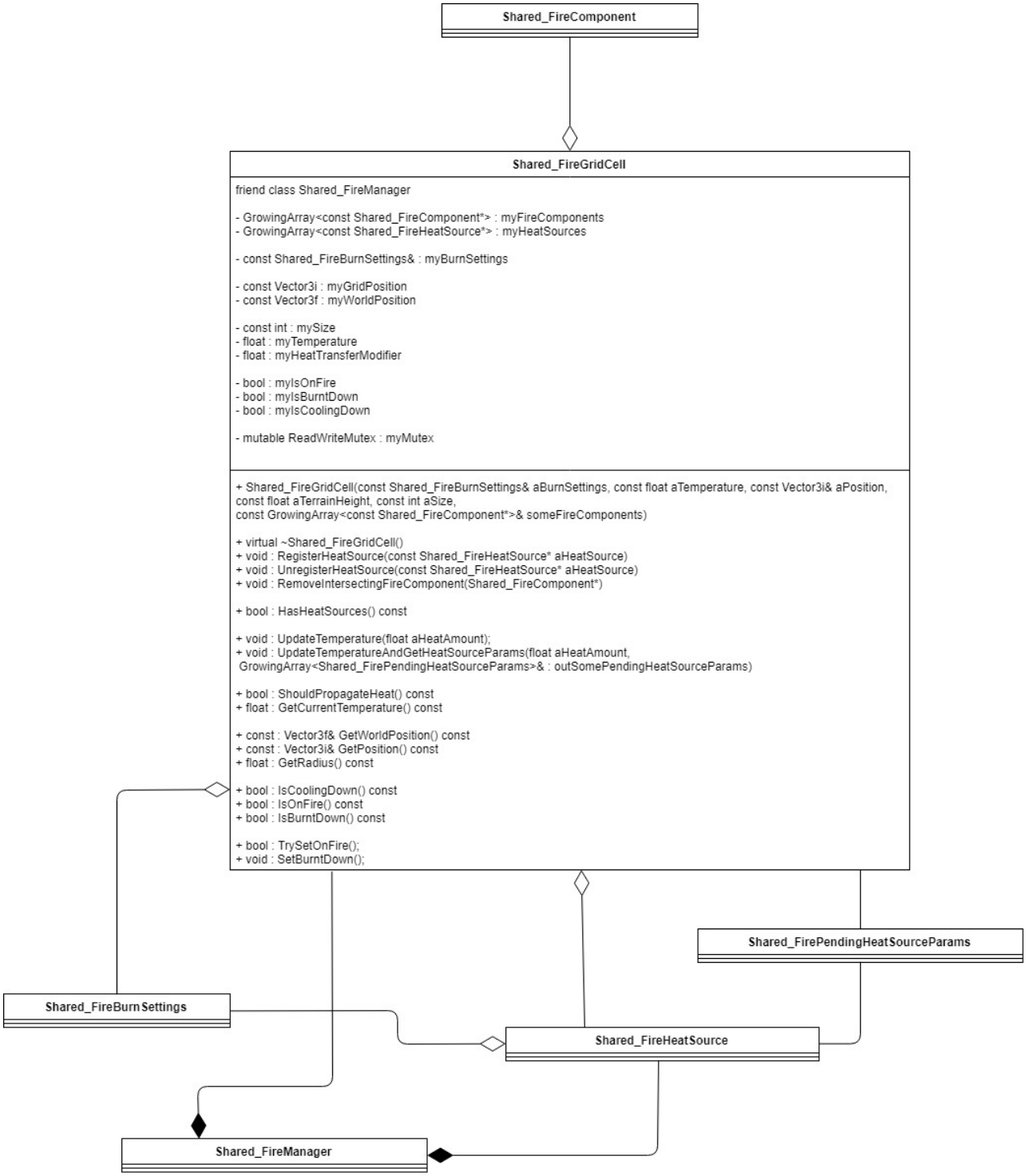

FIGURE 4.21: `FireGridCell` UML

The `RemoveIntersectingFireComponent()` function was added in order to remove items from `myFireComponents` in case a the instance of `FireComponent` an item is pointing to is unloaded. All elements in `myFireComponents` are passed to the constructor as an argument. During the life cycle of a `FireGridCell` instance, no element will be added.

FireHeatSource

As the only *Heat Source* in the system, the `FireHeatSource` class is responsible for propagating heat and fire. The `FireHeatSource` class has two derived classes, `FireCellHeatSource` and `FireObjectHeatSource`. If not explicitly stated otherwise, the term `FireHeatSource` includes base and derived classes. Other classes within the system almost exclusively store instances of these derived classes within pointers to the base class. This is why `myOrigin` provides a way to determine whether the instance pointed to is of the base class or one of the derived classes. `FireHeatSource` instances are not arranged in a grid structure, all instances are stored in `FireManager.myUpdatedHeatSources`. `myBurnSettings.myPropagationTime` defines how long an instance will propagate heat. This is checked by subtracting `myCreationTime` from the current game time and check if the result is smaller or bigger than the `myBurnSettings.myPropagationTime`. `myCreationTime` is initialized with a negative value and is only set in the `SetActive` function, which is called when the instance is added to `FireManager.myUpdatedHeatSources`. The `PropagateHeat()` (15) function updates `myTemperature` and calls UpdateTemperatureAndGetHeatSourceParams() on each `FireGridCell` instance in `myHeatedFireGridCells`. The returned `FirePendingHeatSourceParams` instances are passed to the `FireManager`, which creates `FireHeatSource` instances. The `UpdateCurrentTemperature()` function updates `myTemperature` by multiplying `myBurnSettings.myMinTemperatureChangePerSeconds` with the time of the last frame and either adds it to `myTemperature` or subtracts it. This depends on whether `currentTime - myCreationTime < myPropagationTime / 2` or not.
The `CalculateHeatAmount()` function uses 4.1 for its calculation of the thermal energy transferred to each `FireGridCell`.
`SetInactive()` is a pure virtual function declared in the base class. As the base class does not have an instance of another class as origin, it does not need to manage its internal state. Both derived classes, `FireCellHeatSource` and `FireObjectHeatSource`, implement this function and set the state of `FireCellHeatSource.myFireCell` and `FireObjectHeatSource.myFireState` to `BurntDown` respectively.
The `FillFireHeatSourceCreationParams()` function writes the member values of an `FireHeatSource` instance to an `FireHeatSourceCreationParams` instance. As both derived classes have additional member variables, this functions is overridden.

FireObjectHeatSource The `FireObjectHeatSource` class stores a pointer to a `FireState` instance as its origin (`myFireState`). In

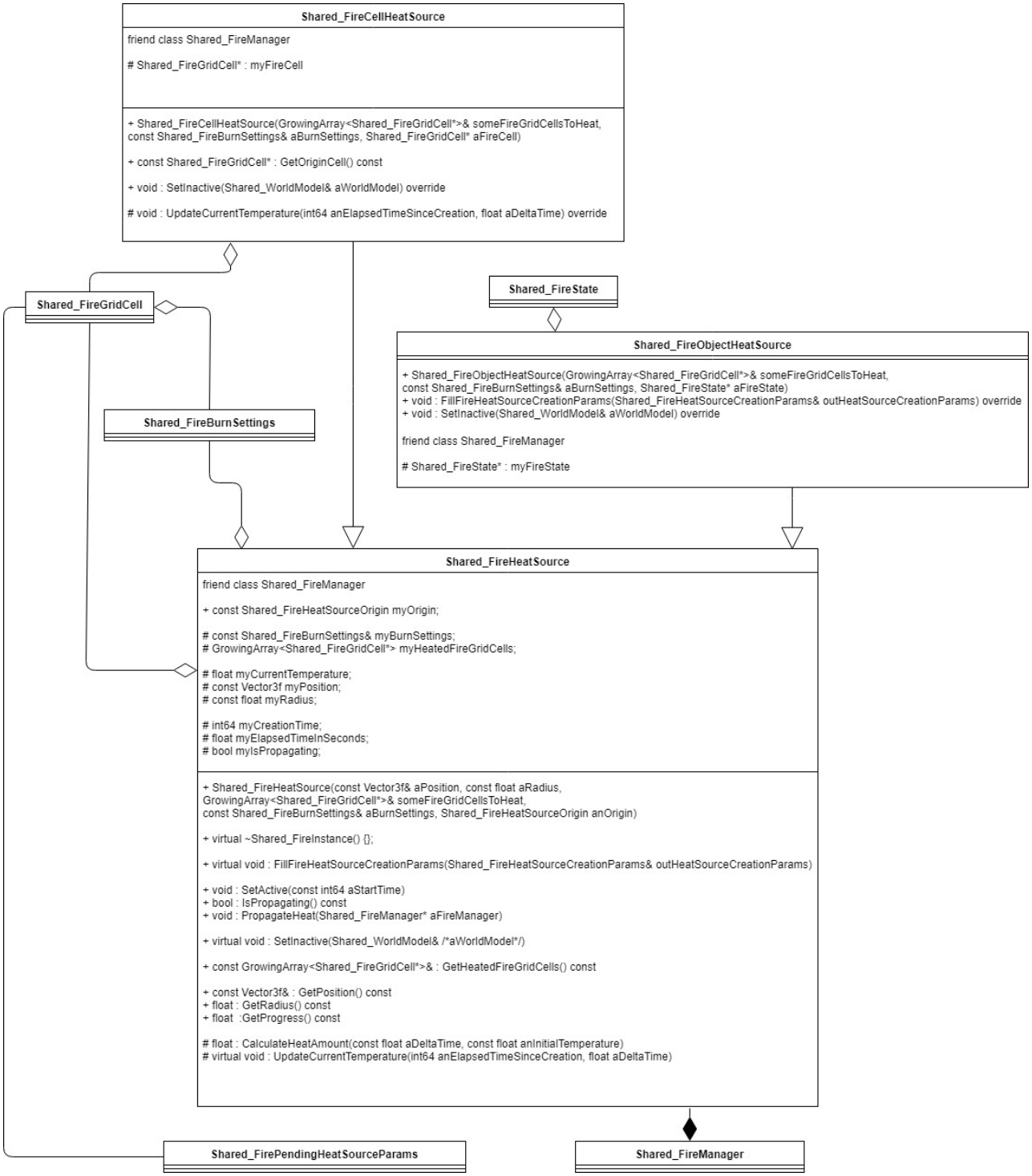

FIGURE 4.22: `FireHeatSource` UML

Function 15: `FireHeatSource:PropagateHeat(FireManager)`

```
Input: aFireManager
Output: No output
if (myIsPropagating == false or myCreationTime < 0)
{
    return
}
get currentTime
elapsedTimeSinceCreation = currentTime - creationTime
if (elapsedTimeSinceCreation > myBurnSettings.myPropagationTime)
{
    myIsPropagating = false
    return
}
UpdateCurrentTemperature(elapsedTimeSinceCreation,deltaTime)
for (each fireGridCell in myHeatedFireGridCells)
{
    heatAmount = CalculateHeatAmount(deltaTime, fireGridCell.myTemperature)
    heatSourceParams =
     fireGridCell.UpdateTemperatureAndGetHeatSourceParams(heatAmount)
    aFireManager.CreatePendingHeatSources(heatSourceParams)
}
```

the overridden function `FillFireHeatSourceCreationParams()`, `myFireComponent.myEntityID` of the `FireState` instance is assigned to the `FireHeatSourceCreationParams.myFireComponentUID` member. The `SetInactive()` function transitions the state of `myFireState` to `BurntDown`. This function is guaranteed to be called before the `FireObjectHeatSource` instance is destroyed.

FireCellHeatSource This derived class of `FireHeatSource` stores a pointer a `FireGridCell` instance as its origin (`myFireCell`). The `FillFireHeatSourceCreationParams()` function did not have to be overridden, because a `FireGridCell` instance can be identified via the position. The position is a member of the base class and written to a `FireHeatSourceCreationParams` instance in the base class function. The `SetInactive()` function sets `FireGridCell.myIsBurntDown` to true and `FireGridCell.myIsOnFire` to false.

4.3.3 Limitations

Although fully functional, there are some known limitations of the presented system in its current form. While flammable objects represented by `FireComponents` can change their entity state to `OnFire`, this system does not handle the visual representation of a single object. Visually, the area around this object is burning while technically, the visual aspect references to the `FireGridCell` instance that created a `FireCellHeatSource` at this position.

While this makes the creation of a lot of `FireComponent` instances obsolete, it introduces a new problem. As the system does not differentiate between ground materials, the system can not prevent fire from spreading under water or on not flammable ground materials.
Furthermore, taking the example of a player using fire arrows to start a propagating fire, the fire does not start to spread at the exact location of the arrow hit. This effect is increasing with a growing cell size, as the initial hit position is snapped to the closest `FireGridCell` instance which will then spawn a `FireCellHeatSource` instance at its center and with it a particle effect.

Chapter 5

Evaluation

The focus of this chapter is the evaluation and comparison of the two approaches presented in this thesis. They were assessed with regard to the requirements defined in chapter 4.1. These were condensed into two aspects: First of all, it will be discussed how both approaches simulate the propagation of fire with regard to realism and credibility. Secondly, performance measurements were executed to test scalability. It has to be noted, that these measurements were only executed for the cell-based approach for reasons discussed in this section.

5.1 Realism

Requirement 1 relates directly to this aspect. While the approaches do not claim to simulate the propagation of fire in a realistic way, they still have to be convincing for the player. Because the project this system was developed for is in a very early stage of development, many systems are expected to be incomplete and objects to be placeholders. The credibility of a game world relies heavily on the visual and auditory aspects, which were not created in the context of this thesis. Therefore, it is impossible to tell if players would consider either of the approaches believable. On evaluation the approaches it is important to remember that the present stage of development, both approaches only provide basic functionality and do not handle special cases. Considering this, the system would have an undefined behavior in those cases. Additionally, all defined parameters need to be setup for objects within the game world. This is crucial for the system as it heavily influences the propagation behavior of objects for both approaches. Proper parameter setup was not part of the scope of this thesis. In its current state, none of the presented approaches meet requirement 1.

5.1.1 Summary

It was not possible to meet the requirements as there was no foundation on which to set up a testing environment in the current phase of the project. Whether or not players feel like the system behaves realistic or as expected needs to be answered when the actual gameplay is defined and players can experience different systems interacting with each other.

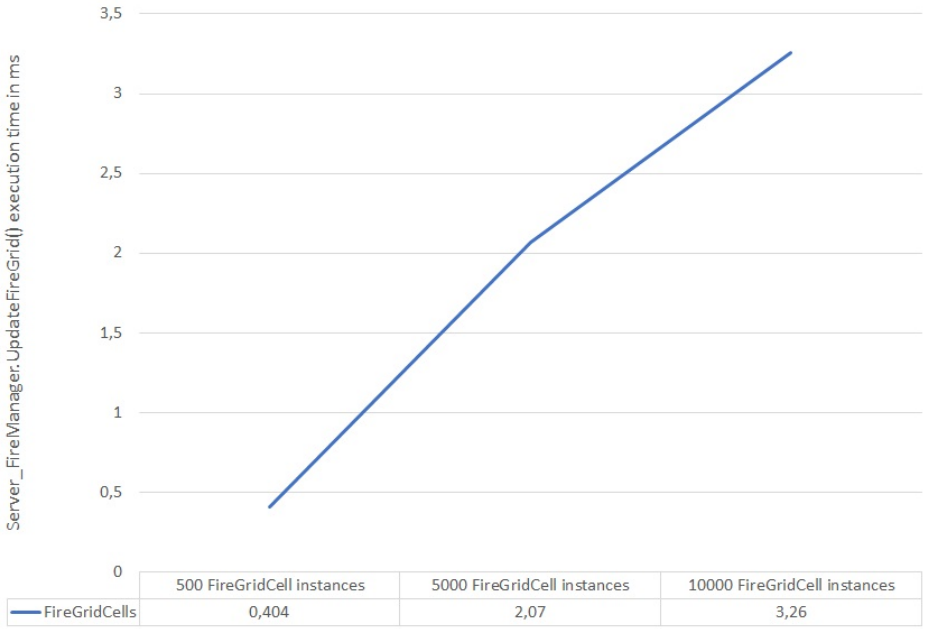

FIGURE 5.1: Performance Evaluation

5.2 Performance

5.2.1 Experimental Set-Up

Server performance is particularly important in an MMOG that can only be played online. A high RTT could also be the result of very high frame times on the server process. Performance measurements were executed on a single workstation equipped with an *Intel Core i7-7700k* processor and 16 gigabytes of RAM (Random Access Memory). The graphics card is not of importance for this measurement as the server process does not display any graphics. For local testing purposes, only one client was connected to the dedicated server. For this performance measurement, the execution time of `Server_FireManager.UpdateFireGrid()` was measured. The performance of this function was measured over multiple frames with 500, 5000 and 10000 `FireGridCell` instances in the game world, regardless of their state. To filter out small frame time jittering, the average execution time of five consecutive frames was computed.

5.2.2 Results

Overall the results (5.1) show a time complexity smaller than `O(n)`. The execution time of the `Server_FireManager.UpdateFireGrid()` on the server process is at 3,26 milliseconds (ms) for 10000 `FireGridCell` instance in the game world. As one of the requirements was to optimize this system for 500 players, 10000 is the equivalent of on average 20 `FireGridCell` instances per player. For keeping a constant frame rate of 10 frames per second, the average frame time should be 100ms or less. With

3,26ms, it takes up exactly *3,26%* of the frame time budget. As an average number of `FireGridCell` instances per player can not be defined yet, it is impossible to say if a total of 10000 `FireGridCell` instances will ever be reached on a single server process.
During these measurements no frame spikes could be observed, which meets requirement 4. Special circumstances made it unnecessary to measure performance for the object-based approach. During regular conducted playtests in the context of the project at *Massive Entertainment*, heavy and frequent frame spikes were discovered when players loaded or unloaded sectors with many `FireComponent` instances attached to entities. It was discovered that this was mainly due to mutex contention when many `FireComponent` were added and removed from the `SpatialGrid` instance. The insertion or deletion of a `FireComponent` in the spatial hash was becoming a more performance intensive task with an increasing resolution. The resolution of the spatial hash is entirely dependent on the chosen cell size.
Decreasing the resolution and thus increasing the size of a cell within the grid caused another problem. The size of the grid cells determines whether a `FirePropagationAABB` instance should grow or not. Since the `FirePropagationAABB` instance essentially merges with another AABB, it grows in every direction, which makes the propagation increasingly inaccurate, resulting in fire being spread in every direction regardless of the direction the burning object is. This made clear that the object-based approach was not usable for this project. Which is also the reason why no performance measuring for this approach was conducted.

5.2.3 Summary

In sum, the results show that it was not possible to tell if most of the performance related requirements were met or not. This is due to project development circumstances and requirements not fit for a very early prototype.

Chapter 6

Conclusion

In this thesis, two approaches for developing a Fire Propagation System for an MMOG were presented. The first chapter provided, background knowledge on MMOGs, their topology, how clients and server in this topology keep the simulation synchronized and on fundamentals of Fire Dynamics. Before the approaches were presented, an overview of relevant functionality concerning the framework was given. Although the concepts were developed without specific framework requirements, the presented implementation of both approaches relied heavily on existing concepts, features and systems within the used framework. It was discovered, that the object-based approach was not feasible for the given framework, as even in tests with a small number of players, frame spikes were repeatedly discovered.
Requirement 1 (Fire spreads in a way within the game world that players would expect) could not be evaluated as many parts facing players, which were mostly of a visual nature, were still in a very early stage This posed difficulties for testers in judging whether the system behaved as expected. For the implementation of both approaches, existing systems were used heavily. This made communication with other, already existing, systems much easier and fulfilled requirement 2. Because of reasons explained in 5.2.2, requirement 3 (The system has to be optimized to run for 500 clients connected to a single dedicated server) could not be fulfilled by the object-based approach. On the other hand, the performance measuring 5.2.2 for the cell-based approach shows that the system can handle a very high number of burning objects. Nonetheless, just like the project this system was developed in a very early stage of development, in which realistic end user conditions were not given. As mentioned before, requirement 4 (There should be no frame time spikes on the server) could not be fulfilled by the implementation of the object-based approach, while the cell-based approach did not show any significant spikes in the performance measuring. Given the dedicated server topology that was assumed for the presented approaches, requirement 5 (The system should be server authoritative) ensures that local client changes can not override the simulation state of the server and thus for every other connected client. To meet this requirement, a fire can only be started on the server. Additionally, the server communicates all state changes to its clients. However, local changes can alter the way the Fire Propagation System behaves without the server correcting it. While this results in a visual inconsistency,

it has no other impact on the game experience. The fulfilling of requirement 6 (The propagation has to be deterministic) was ensured by avoiding any kind of time dependent, or random values.
While the object-based approach presented in this thesis will not be developed any further, the implementation of the cell-based approach was a promising start, as most requirements could be fulfilled. It provides a solid base functionality that can be expanded and tailored for any project-specific requirements.

Bibliography

[1] David Aldridge. *I Shot You First: Networking the Gameplay of HALO: REACH*. 2011. URL: http://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking (visited on 06/11/2018).

[2] Boreal Games. *Understanding Component-Entity-Systems*. 2013. URL: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/understanding-component-entity-systems-r3013/ (visited on 05/18/2018).

[3] Carl's Jr. *Spatial Hashing in C++, Part 1*. 2016. URL: http://www.sgh1.net/posts/spatial-hashing-1.md (visited on 06/15/2018).

[4] Fábio Reis Cecin et al. "FreeMMG: a hybrid peer-to-peer and client-server model for massively multiplayer games". In: *NetGames '04: Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games* (2004), p. 172. DOI: http://doi.acm.org/10.1145/1016540.1016567.

[5] Takuhiro Dohta, Hidemaro Fujibayashi, and Satoru Takizawa. *Change and Constant: Breaking Conventions with "The Legend of Zelda: Breath of the Wild"*. San Francisco, 2017. URL: https://www.gdcvault.com/play/1024562/Change-and-Constant-Breaking-Conventions.

[6] Dynamight Studios. *Fractured: The Dynamic MMO*. 2018. URL: https://fracturedmmo.com (visited on 02/10/2018).

[7] Cliff Edwards. *PC Games King Seeks to Dethrone Sony, Microsoft Consoles*. 2013. URL: https://www.bloomberg.com/news/articles/2013-09-26/pc-games-king-seeks-to-dethrone-sony-microsoft-consoles (visited on 02/24/2018).

[8] Sergey Galyonkin. *Steam Spy : Games released in previous months*. 2018. URL: https://steamspy.com/year/ (visited on 02/17/2018).

[9] Josh Glazer and Sanjay Madhav. *Multiplayer Game Programming: Architecting Networked Games*. 1th Editio. Addison-Wesley Professional, 2015, p. 384.

[10] Improbable Worlds Limited. *SpatialOS Technical Breakdown*. 2017. URL: https://improbable.io/games/tech.

[11] Interactive Software Federation Of Europe. *GameTrack Digest : Quarter 3 2014*. Tech. rep. 2014. URL: https://www.isfe.eu/sites/isfe.eu/files/gametrack_european_summary_data_2014_q3.pdf.

[12] Interactive Software Federation Of Europe. *GameTrack Digest : Quarter 3 2015*. Tech. rep. 2015. URL: https://www.isfe.eu/sites/isfe.eu/files/gametrack_european_summary_data_2015_q3.pdf.

[13] Interactive Software Federation Of Europe. *GameTrack Digest : Quarter 3 2016*. Tech. rep. 2016. URL: https://www.isfe.eu/sites/isfe.eu/files/gametrack_european_summary_data_2016_q3.pdf.

[14] Interactive Software Federation Of Europe. *GameTrack Digest : Quarter 3 2017*. Tech. rep. 2017. URL: https://www.isfe.eu/sites/isfe.eu/files/gametrack_european_summary_data_2017_q3.pdf.

[15] Jassin Kessing, Tim Tutenel, and Rafael Bidarra. "Designing Semantic Game Worlds". In: *Proceedings of the The third workshop on Procedural Content Generation in Games - PCG'12* (2012), pp. 1–9. DOI: 10.1145/2538528.2538530. URL: http://dl.acm.org/citation.cfm?doid=2538528.2538530.

[16] Jean-Francois Levesque. *Far Cry : How the Fire Burns and Spreads*. 2012. URL: http://jflevesque.com/2012/12/06/far-cry-how-the-fire-burns-and-spreads/ (visited on 10/03/2018).

[17] Adam Martin. *Entity Systems are the future of MMOG development – Part 2*. 2007. URL: http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/ (visited on 05/22/2018).

[18] Massive Entertainment. *The Snowdrop Engine*. 2018. URL: https://www.massive.se/project/snowdrop-engine/#about (visited on 06/08/2018).

[19] Jessica Mulligan. *Developing Online Games: An Insider's Guide*. 2003, p. 495. ISBN: 1592730000.

[20] National Institue of Standards and Technology. *Fire Dynamics*. 2010. URL: https://www.nist.gov/<front>/fire-dynamics (visited on 05/16/2018).

[21] Ltd. Nintendo Co. *The Legend of Zelda : Breath Of The Wild*. 2017. URL: https://www.zelda.com/breath-of-the-wild/ (visited on 06/11/2018).

[22] Robert Nystrom. *Game Programming Patterns*. 1 edition. Genever Benning, 2014, p. 354. ISBN: 978-0990582908. URL: http://gameprogrammingpatterns.com/component.html.

[23] SuperData Research. *The MMO & MOBA Games Market Report*. 2016. URL: https://www.superdataresearch.com/market-data/mmo-market/ (visited on 02/24/2018).

[24] The Concord Consortium. *Heat & Temperature*. URL: http://energy.concord.org/energy2d/ht.html (visited on 05/16/2018).

[25] Ubisoft Entertainment. *Third Quarter Sales Report*. Tech. rep. February. 2018.

[26] Unity Technologies. *Entity Component System (ECS)*. 2018. URL: https://unity3d.com/unity/features/job-system-ECS (visited on 05/18/2018).

[27] Unity Technologies. *Unity Documentation - Transform*. 2018. URL: https://docs.unity3d.com/Manual/class-Transform.html (visited on 06/07/2018).

[28] Valve Corporation. *LIONSGATE LAUNCHES FILMS ON STEAM'S GLOBAL DIGITAL DISTRIBUTION PLATFORM*. 2016. URL: http://store.steampowered.com/news/21534/ (visited on 02/24/2018).

[29] Valve Corporation. *Steam & Game Stats*. 2018. URL: http://store.steampowered.com/stats/ (visited on 02/10/2018).

[30] David Xicota. *Lag compensation techniques for multiplayer games in realtime*. 2016. URL: http://www.gamedonia.com/blog/lag-compensation-techniques-for-multiplayer-games-in-realtime (visited on 06/11/2018).